A Testbed for Protocol Analysis for the Internet of Things

Edward Sereko Younge

Master's Thesis

Aalto University – School of Electrical Engineering, Finland

Department of Communications and Networking (Comnet)

May 2012

edward.younge@aalto.fi

Student Number: **85070J**

Instructor:

Supervisor: **Joerg Ott**

Completed: **26 May, 2012**

Abstract

In future smart environments, wireless sensor networks will play a key role in sensing, collecting, and disseminating information about environmental phenomena. These are made possible by the availability of sensors that are smaller, cheaper and intelligent. In addition, the sensors have wireless interfaces with which they communicate with one another in a network. However, these sensors have limited processing resources, memory and power. Due to such limitations, the design of a wireless sensor network depends heavily on the application and associated factors such as the environment, the objectives of the application design, cost, hardware, and system constraints. This thesis investigates the difference(s) between the PUSH and PULL protocols by the use of a test-bed in different topologies of a wireless sensor network, in the “Internet of Things” concept. Also, the better protocol, if applicable, is selected.

Table of contents

List of acronyms

CoAP – Constrained Application Protocol

UDP – User Datagram Protocol

6LoWPAN – ipv6 over Low power Wireless Personal Area Network

URI – Uniform Resource Identifier

TLV – Type-Length-Value

RTT – Round Trip Time

REST – Representational State Transfer

OC – Option Count

HTTP – Hyper Text Transfer Protocol

RPL – ipv6 Routing Protocol for Low power and lossy networks

LLN – Low power and Lossy Network

ND – Neighbor Discovery

DAG – Directed Acyclic Graph

DODAG – Destination Oriented Directed Acyclic Graph

OF – Objective Function

PCB – Printed-Circuit-Board

1. Introduction

In future smart environments, wireless sensor networks will play a key role in sensing, collecting, and disseminating information about environmental phenomena. [1] These are made possible by the availability of sensors that are smaller, cheaper and intelligent. In addition, the sensors have wireless interfaces with which they communicate with one another in a network. However, these sensors have limited processing resources, memory and power. Due to such limitations, the design of a wireless sensor network depends heavily on the application and associated factors such as the environment, the objectives of the application design, cost, hardware, and system constraints. [2]

Sensor networks have been for a while subject to research [17, 18, 19, 20, 21] and that today's technology can bring this closer to reality. Also, sensor networks are no longer used in isolation, but seen as part of the Internet, which gets us to the "Internet of Things" concept.

This thesis studies the difference(s) between the PUSH and PULL protocols by the use of a test-bed in different topologies of a wireless sensor network (WSN), in the "Internet of Things" concept.

This is to understand the behavior of the protocols and/or the technologies used in the "Internet of Things", one element of the future of the Internet.

1.1Motivation

Sensor networks are usually in closed areas and thereby making them restricted to certain locations, thus do not contribute to the concept of a "Global Village" as the world is referred to in recent years because of the "Internet". Hence, it makes sense to have sensor networks in the "Internet of Things" concept. Also, "Internet of Things" is seen to be one element of the future of the Internet, sensor network for that matter. This thesis aims at investigating how well the protocols, currently used in the "Internet of Things", perform.

1.2 Research Question

In general, a sensor network comprises of sensors and actuators, and then components for collecting data and giving commands. This leads to your communication needs and the fact that you use different types of protocols, push and pull. The PULL and PUSH protocols are the two protocols used in wireless sensor networks, and one may choose either of the two

depending on the type of application or the data gathering method deemed suitable. However, both can be used in any wireless sensor network. The purpose of this thesis is therefore to compare the two and select the better protocol, if possible. Also, the real behavior of the protocols in the various networks will be studied and so no simulation will be used. The research question is the following:

What are the main differences between the PULL and PUSH protocols with respect to network behavior? Which protocol has a better performance?

1.3 Scope and structure of the thesis

This thesis focuses on the differences between the PUSH protocol (regular UDP) and the PULL protocol (CoAP) with respect to network behavior due to changes in traffic rates.

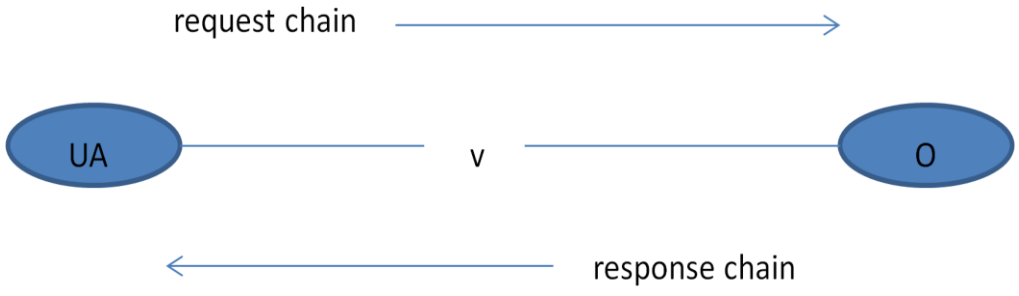
The structure of the thesis is the following: Chapter 2 contains literature review of the concepts/protocols used in this thesis and these are HTTP, LLN, REST, CoAP and RPL. The third chapter gives an overview of the research carried out with respect to the experiments performed, the evaluation methodology used, and the building and use of a testbed. Chapter 4 explains the testbed and the fifth chapter shows the different test cases used in this thesis. The sixth chapter presents the results of the test cases outlined in the previous chapter. Chapter 7 discusses the results obtained in the previous chapter, and the last chapter gives the findings of the research and the conclusions drawn from them.

2. Background

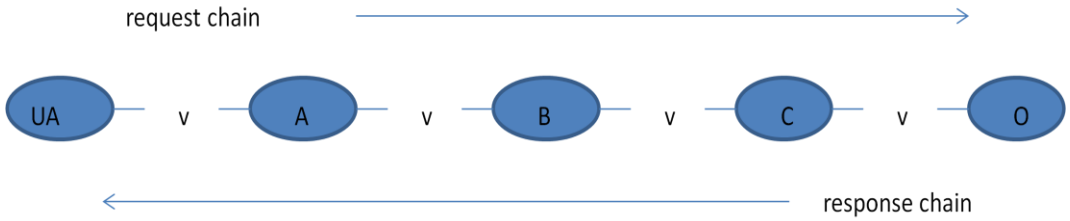
This chapter gives the theoretical background of the main concepts used in this work. These are the REST architecture, the CoAP protocol and the RPL protocol. The web environment using HTTP (Hyper Text Transfer Protocol), is used for distributed, collaborative and hypermedia information systems on the application level. Practical information systems need more functionality than simple retrieval, in addition to search, front-end update and annotation. HTTP supports an open-ended set of methods and headers that show the purpose of a request. It builds on the discipline of reference provided by the Uniform Resource Identifier (URI), as a location (Uniform Resource Locator) or as a name (Uniform Resource Name), for showing the resource to which a method is to be applied. The messages are passed in a way identical to that used by the Internet mail as specified by the Multipurpose Internet Mail Extensions (MIME). [5] MIME allows for textual message bodies in character sets other than US-ASCII, an extensible set of different formats for non-textual message bodies, multi-part message bodies and textual header information in character sets other than US-ASCII. [6]

Furthermore, HTTP is used as a generic protocol for interaction between user agents and proxies/gateways to other Internet systems, in addition to those supported by SMTP (Simple Mail Transfer Protocol), NNTP (Network News Transfer Protocol), FTP (File Transfer Protocol), Gopher and WAIS (Wide Area Information Server) protocols. In this way, Hyper Text Transfer Protocol allows basic hypermedia access to resources available from different applications. HTTP also uses the request/response model of communication in which a client sends a request to a server using a request method, URI (Uniform Resource Identifier), and protocol version preceding a MIME-like message that contains, among other things, client information. The server then replies with a status line, in addition to the message's protocol version and a success or error code, preceding a MIME-like message containing, among other things, server information. [5]

Below is a simple example of an HTTP interaction initiated by a user agent (UA e.g. client) through a single connection (v) to an origin server (O).

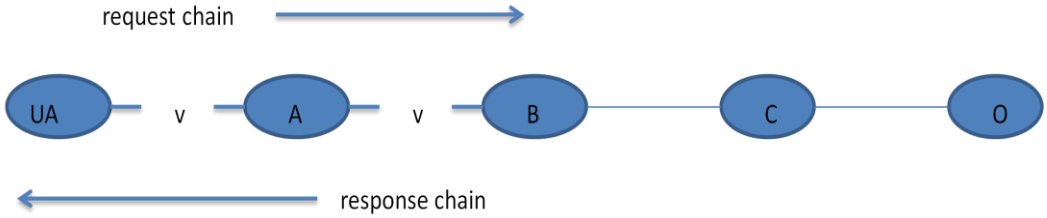


Also, below is a more complex example in which there exist intermediaries between the user agent and the origin server.



An intermediary can be a proxy, gateway or tunnel. The proxy is basically a forwarding agent that receives requests for a URI in its absolute form, rewrites all or part of the message and forwards the reformatted request toward the identified server in the URI. On the other hand, the gateway is a receiving agent that acts as a layer other server(s) and translates the requests to the underlying server's protocol, if necessary. The tunnel acts as a relay point between two connections without modifying the messages and this is usually used when there is a firewall between the two parties communicating.

Moreover, a party to a communication that does not act as a tunnel could employ an internal cache for handling requests and this may result in a shortened request/response chain, as shown below.



In the example above, "B" has a cached copy of an earlier response from "O", through C, for a request that has not been cached by the user agent or "A". [5]

Other terminologies used in HTTP are resource, variant, entity, representation, content negotiation, first-hand, explicit expiration time, heuristic expiration time, age, freshness lifetime, stale, semantically transparent, validator, upstream/downstream and inbound/outbound. A resource is network data object or service that can be identified by a URI. An entity is the information transferred as the payload of a request or response whereas a representation is an entity included with a response that is subject to content negotiation.

Content negotiation, on the other hand, is the mechanism for selecting the appropriate representation when servicing or satisfying a request. Each representation of a resource is known as a variant. A response is first-hand if it comes directly and without unnecessary delay from the origin server. Explicit expiration time is the time at which the origin server intends that an entitity should no longer be returned by a cache without further validation whereas heuristic expiration time is an expiration time assigned by the cache when no explicit expirationtime is available. Age of a response, on the other hand, is the time since it was sent by, or successfully validated with, the origin server whereas freshness lifetime is the length of time between the generation of a response and its expiration time. Also, a response is fresh if its age has not yet exceeded its freshness lifetime whereas a response is stale if its age has passed its freshness lifetime. A cache is said to be "semantically transparent", with respect to a particular response, if its use affects neither the requesting client nor the origin server, except to improve performance. A validator is also a protocol element used to find out whether a cache entry is an equivalent copy of an entity. Last but not least, upstream and downstream indicate the flow of a message (all messages flow from upstream to downstream) whereas inbound and outbound refer to the request and response paths for messages (inbound indicates "traveling toward the origin server" whereas outbound refers to "traveling toward the user agent"). [5]

However, in a restricted environment, such as LLN (Low power and Lossy Network), nodes or user agents and origin servers must use resources efficiently. LLNs are comprised of several embedded devices with limited power, memory and processing resources. These devices are interconnected by a variety of links, such as Bluetooth, IEEE 802.15.4, Low Power WiFi, wired or other low power PLC (PowerLine Communication) links. These networks are also transitioning to an end-to-end IP-based solution in order to prevent the problem of non-interoperable networks interconnected by protocol translation gateways and proxies.

The four main distinguishing characteristics of LLNs are:

- LLNs optimize for energy saving in most situations
- Point to multipoint is a typical traffic pattern in LLNs
- LLNs are employed over link layers having restricted frame-sizes
- Many LLN nodes or devices do not have resources to waste

These are the main reasons why new protocols are needed to satisfy these requirements. Thus, the use of CoAP (Constrained Application Protocol) and RPL (IPv6 Routing for Low power and lossy networks) based on the REST (Representational State Transfer) architecture. [7]

2.1 REST

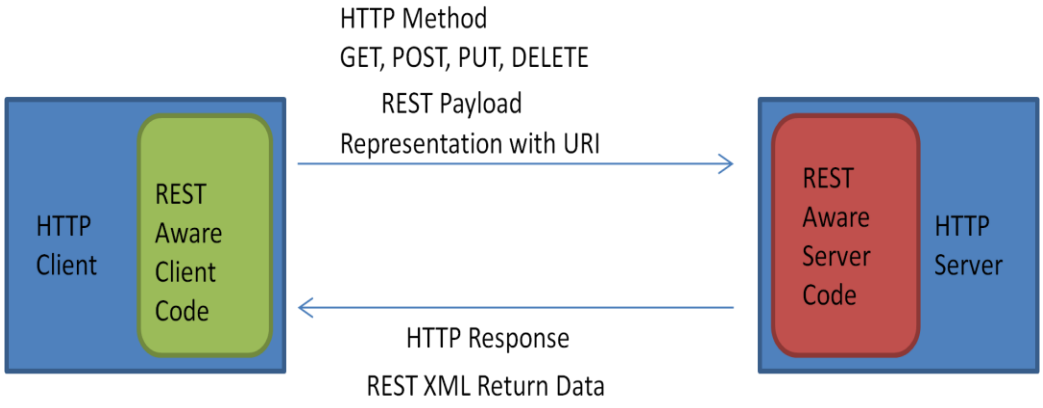
REpresentational State Transfer (REST) is an architectural style for networked systems and not protocol-dependent, though most practical implementations are built on HTTP. Moreover, REST is not a standard but prescribes the use of standards such as XML (for resource representation in Extensible Markup Language which is both human-readable and machine readable), TEXT/XML (for content type in text form encoded in Extensible Markup Language), HTTP and URL.

2.1.1 Design Principles of a REST-based System

The communication between a client and a server must be stateless, in the sense that information known to the server must not be used but rather a request from a client must include all the information needed by the server to respond appropriately. Also, there should be a uniform interface that supports state transfer and comprising of a constrained set of properly defined operations (such as HTTP methods GET, PUT, POST and DELETE) and a constrained set of content types (such as TEXT/XML and IMAGE/JPEG. Furthermore, there should be Client-server pull interaction with client pull representations. Also, resources should be uniquely named by the use of URI (Uniform Resource Identifier). In addition, there should be layered, interconnected resource representations by the use of URLs (Uniform Resource Locators) in order to make progression through states by clients possible. Last but not least, there should be cacheable responses in order to make the network operate efficiently.

2.1.2 REST Architecture

Below is a REST-based architecture for client-server interaction.



As shown above, HTTP is used as the uniform interface and the resource representations are shared over this interface. On the client side, the REST aware client code is usually part of a web page, loaded from a web server. This client is commonly written using JavaScript and embedded into HTML for manipulating resources. The client side code also parses and acts upon data returned from the server side based on the logic of the application. Furthermore, the architecture above shows how different underlying standards such as HTTP, URI, HTML and XML are used in supporting a REST based architecture. [8]

2.1.3 Advantages and Disadvantages of a REST based Architecture

Below are some of the commonly known advantages of REST. [8]

It offers possibilities for the development of thin clients by the use of less client code. Also, it does not require explicit resource discovery mechanism due to hyperlinking. Furthermore, it is a scalable architecture due to its stateless nature. The use of caching makes the network operate efficiently with fast response times. Benefits of software versioning include support of document type evolution such as HTML and XML without affecting backward or forward compatibility. The use of resource extensions makes it possible to add new content types without affecting existing and legacy content types.

Also, below are some of the commonly known disadvantages of REST. [8]

It faces technical challenges for real time asynchronous events to a thin client or browser-based application by using HTTP as a uniform interface. Also, managing URI Namespace can be cumbersome. It does not have supporting software tools. Network performance can be affected negatively by encouraging more frequent client-server requests and responses.

2.2 CoAP

Constrained Application Protocol (CoAP) [1] is a specialized transfer protocol for the web and used by constrained networks and nodes in machine-to-machine applications like control of energy consumption through smart energy metering and automated systems. Limited processing power and storage space are some of the characteristic or features of constrained nodes. Constrained networks are usually lossy networks with high rates of packet error and a limited throughput of up to tens of kilobits per second. 6LoWPAN (IPv6 over Low power Wireless Personal Area Network) is a typical example of such a network.

CoAP has an interaction model identical to the client-server model of interaction of HTTP (Hyper Text Transfer Protocol) but the two low power interacting machines or devices require a CoAP implementation in both ends of the communication or interaction. The

client first sends a resource request which is represented by a URI (Uniform Resource Identifier) on the server through method codes similar to that of HTTP. A response from the server also has a response code identical to that of HTTP and may come along with a resource representation. The response codes of CoAP are usually represented as “2.xx”, “4.xx” or “5.xx” whereas those of HTTP are “2xx”, “4xx” or “5xx”. As a result, one could do CoAP-to-HTTP mapping or HTTP-to-CoAP mapping.

CoAP depends on the REpresentational State Transfer (REST) architecture of the web.

However, CoAP interaction is asynchronous through UDP and logically done via a layer of messages supporting optional reliability with the use of exponential back-off. Moreover, CoAP has four message types and they could be requests or responses depending on the method codes and response codes they may carry. The message types are “Confirmable”, “Non-Confirmable”, “Acknowledgement” and “Reset”. Also, CoAP is a single protocol having messaging and requests or response as features of its header. Figure 1 below shows the abstract layering of CoAP.

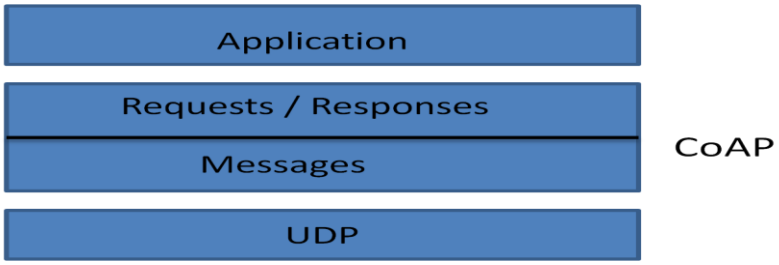


Figure 1: Abstract layering of CoAP [1]

2.2.1 Protocol Operation Overview

The messaging model of CoAP has a short fixed/length binary header of 4 bytes which may precede compact binary options and a payload. The used format is shared between requests and responses. Detection of duplicates and optional reliability are realized by the inclusion of a ”Message ID” which is present in every message. Anytime a message is marked as “Confirmable” (CON), then reliability is guaranteed with retransmission default timeout and exponential back-off between retransmissions until the sender gets an Acknowledgement (ACK) from the receiver carrying the same “Message ID”. This is shown in Figure 2 below.

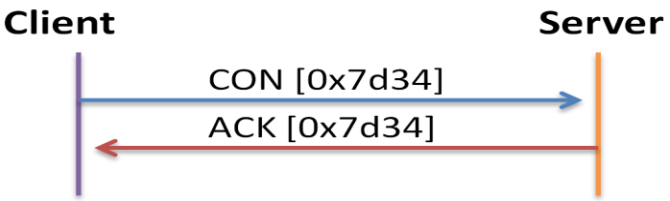


Figure 2: Reliable Message Delivery [1]

However, in the event that the receiver cannot handle the "Confirmable message", the receiver then sends a "Reset message" (RST) to the sender.

Furthermore, a message could be marked as "Non-confirmable", meaning that reliable delivery is not needed and such a case could be the sending of single measurement data from a sensor in a frequent rate. Figure 3 below is such an example.

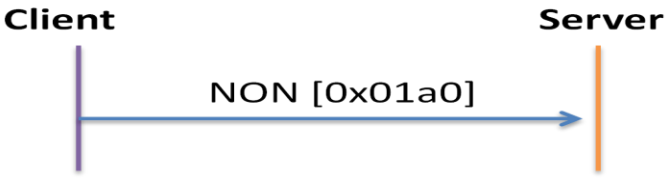


Figure 3: Unreliable Message Delivery [1]

A CoAP message is a request if it contains a "method code" and a response if it contains a "response code". Any request marked as "Confirmable" (CON) gets an immediate response from the receiver in an "Acknowledgement" (ACK) and this reaction is referred to as a "piggy-backed response", provided the response is available. Figure 4 below shows two examples of this mechanism using a simple "GET" request.

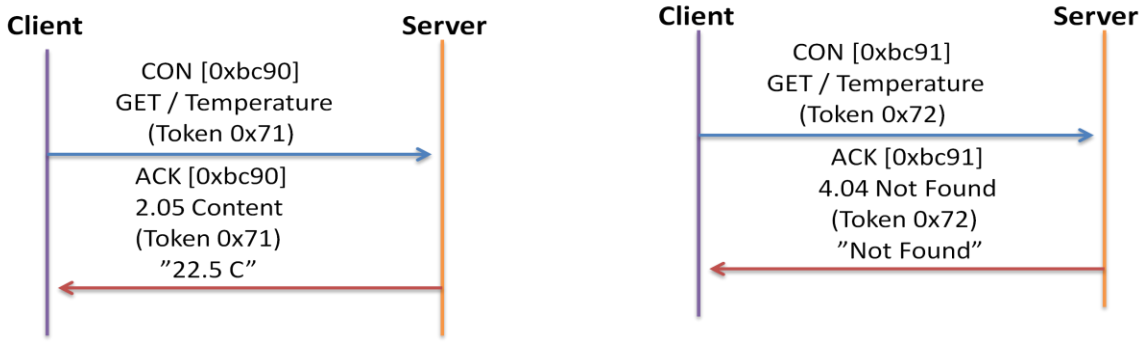


Figure 4: Two GET requests with Piggy-backed responses, one successful, one not found [1]

On the other hand, if the response is not immediately available, the server sends an empty "Acknowledgement message" (ACK) to let the client stop retransmitting the request.

However, immediately the response becomes available, the server then sends it using a new "Confirmable message" (CON), expecting an "ACK" from the client and this makes it a "separate" (second) response to the original request as depicted in Figure 5 below.

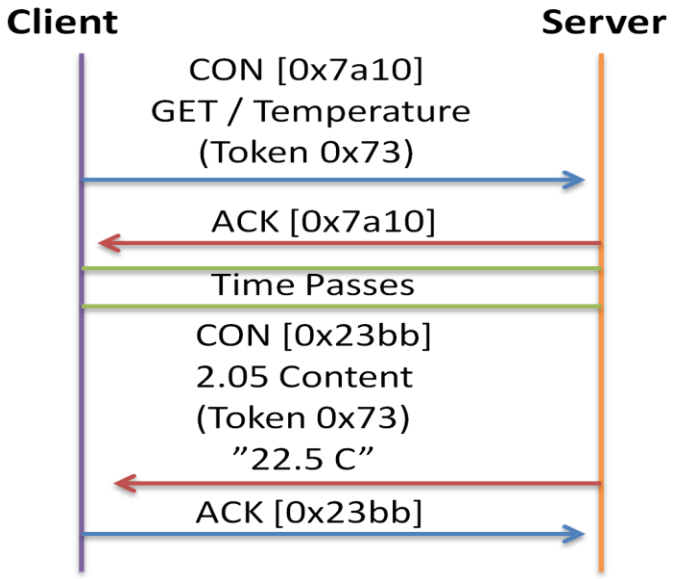


Figure 5: A GET request with a Separate response [1]

Moreover, a "Non-confirmable" message also uses the aforementioned "separate" response mechanism as displayed in Figure 6 below.

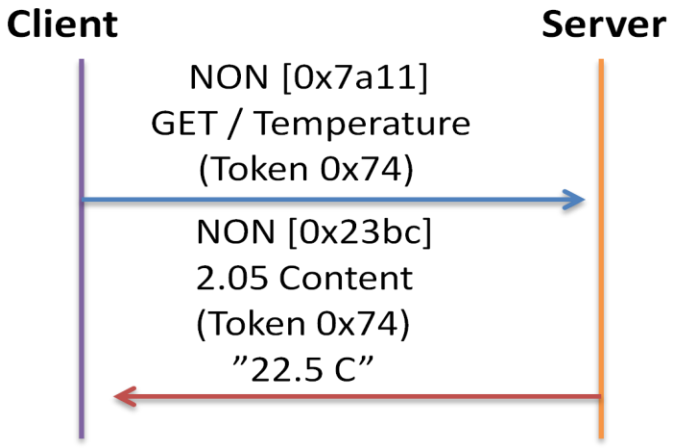


Figure 6: A NON request and response [1]

In addition, caching of responses is supported in CoAP by the use of freshness and validity information included in responses, so as to efficiently satisfy requests. Furthermore, CoAP supports the proxying of requests on behalf of another end-point and this reduces traffic in the network, improves performance, makes accessing resources of sleeping devices

possible and secures the devices. In proxying, the URI of the resource to request is contained in the request, with the destination IP address set to that of the proxy.

2.2.2 Message Format

The message format of CoAP is in binary form and made up of a fixed-sized "Header" succeeded by options in "Type-Length-Value" (TLV) format and the payload, with the number of options being determined by the "Header". The CoAP payload occupies the bytes after the options, if any, and the length computed from the datagram length. This is shown in Figure 7 below.

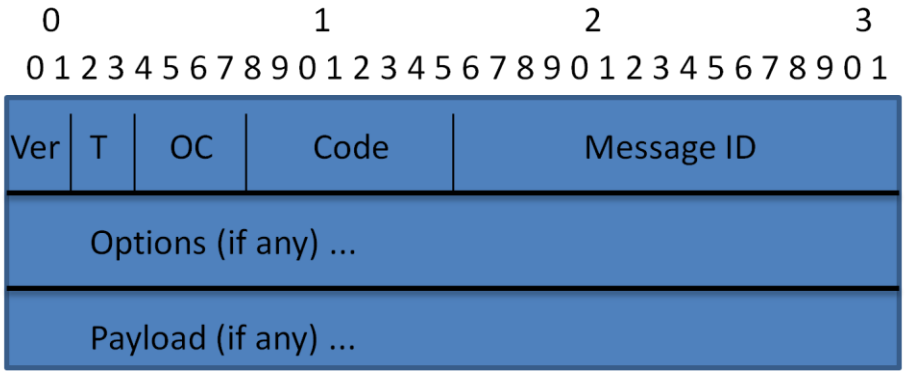


Figure 7: CoAP Message Format [1]

The first field, "Version" (Ver), is a 2-bit unsigned integer data-type which denotes the CoAP version number and this is set to 1 for this specification.

The second field, "Type" (T), is also a 2-bit unsigned integer data-type which signifies if the message is "Confirmable" with value "0", "Non-confirmable" with value "1", "Acknowledgement" with value "2" or "Reset" with value "3".

The third field, "Option Count" (OC)", is a 4-bit unsigned integer showing the number of options which follow the header and if set to "0", signifies that no options are included and if there is any payload it comes just after the header.

The fourth field, "Code", is an 8-bit unsigned integer for signifying if a request message with number range "1 – 31", a response message with number range "64 – 191", or an empty message with number "0".

The fifth field, "Message ID", is a 16-bit unsigned integer for detecting message duplication, matching "Acknowledgement", "Reset" and "Confirmable" message types.

Messages to multicast destination addresses are also supported by the CoAP protocol and the messages are “Non-confirmable”, so as to avoid or reduce congestion. Furthermore, the “exponential back-off” mechanism is used to control congestion.

The compositions of a CoAP request are the method applied to the resource, the resource identifier, the payload, Internet media type if applicable and optional request meta-data. “GET”, “POST”, “PUT” and “DELETE” are the common methods supported by CoAP.

The retrieval of a representation for information corresponding to the resource identified by a request URI is done through the “GET” method. To create or update a resource, the “POST” method is used by enclosing the representation in the request. The “PUT” method, similar to the “POST” method, also uses the enclosed representation to create or update a resource identified by the request URI. A resource identified by the request URI could also be removed or deleted through the use of the “DELETE” method.

On the other hand, a response to a request is identified by a matched client-generated token and a “Response Code”, with a number maintained in the “CoAP Response Code Registry”, set in the “Code” field of the CoAP header. The structure of a “Response Code” is depicted in Figure 8 below.

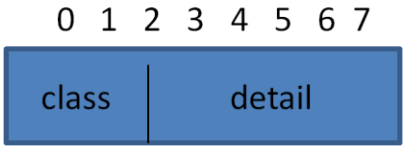


Figure 8: Structure of a Response Code [1]

In Figure 9 above, the class of the response is defined by the upper three bits of the 8-bit Response code number and the classes are three in number. The Class numbered “2” signifies “Success”, “4” signifies “Client Error” and “5” signifies “Server Error”.

2.2.3 Options

Moreover, a single or multiple options could be included in requests and responses. The set of options supported by CoAP are “Content-Type”, “ETag”, “Location-Path”, “ Location-Query”, “Max-Age”, “Proxy-Uri”, “Token”, “Uri-Host”, “Uri-Path”, “Uri-Port”, “Uri-Query”, “Accept”, “If-Match” and “If-None-Match”.

“Token” simply matches a response to a request. “Uri-Host”, “Uri-Path”, “Uri-Port” and “Uri-Query” indicate the target resource of a request to a CoAP server. Specifically, “Uri-Host” indicates the Internet host of the resource being requested, “Uri-Path” shows a segment of the absolute path to that resource, “Uri-Port” denotes the port number of the

resource and “Uri-Query” indicates an argument parameterizing that resource. “Proxy-Uri” is for making the request to the proxy. “Content-Type” shows the representation format of the message payload. “Accept” shows that the included media types are acceptable for the client in the order preference in which they appear. “Max-Age” represents the maximum time a cached response may remain fresh. “ETag” gives the current value of the entity-tag for the enclosed representation of the target resource. “Location-Path” and “Location-Query” show the absolute path URI specifying the location of a resource. “If-Match” is used for updating a resource by matching the “ETag” value, and “If-None-Match” is used for creating a resource in the case whereby the “ETag” value is unmatched.

A Payload may be present in a request if the method is “PUT” or “POST” and in a response if the response is “Content” or “Error”.

A “2.xx” response code signifies “Success”, a “4.xx” response code indicates “Client Error” and a “5.xx” response code denotes “Server Error”.

2.2.4 URI scheme

Resources are identified and located by the help CoAP URIs and the URIs are categorized under two schemes: “coap” and “coaps”. The “coap” scheme is the one without security whereas “coaps” has a security feature. Below is how a “coap” URI looks like.

“coap:” “//” host [“:” port] path-abempty [“?” query]

Below is also how a “coaps” URI looks like.

“coaps:” “//” host [“:” port] path-abempty [“?” query]

In addition, a default port number of “5683” must be supported by a CoAP server during resource discovery and access to other resources. Moreover, a 6LoWPAN node running a CoAP server should support ports in the 61616-61631 compressed UDP port range or space.

2.2.5 Protocol Constants

Furthermore, CoAP has defined constants: the RESPONSE_TIMEOUT is 2 seconds, RESPONSE_RANDOM_FACTOR is 1.5 and MAX_RETRANSMIT is 4.

2.2.6 Security Considerations

In terms of security, CoAP could be secured through different modes of DTLS and IPSec. If DTLS is disabled, then it is termed "NoSec" (No Security) and therefore IPSec Encapsulating Security Payload could be used to secure CoAP. On the other hand, if DTLS is enabled, then it could be the use of "PreSharedKey", "RawPublicKey" or "Certificate". [9]

2.3 RPL

IPv6 Routing Protocol for Low power and Lossy Networks (RPL) [1] is a routing protocol which satisfies the requirements of Low power and Lossy Networks (LLNs). LLNs are made up of nodes constrained in terms of limited memory/storage, processing power and energy, if they are being powered by dry cells (batteries). Moreover, the connections between routers in these networks are lossy with low data and packet delivery rates. Furthermore, the traffic patterns could be point-to-point (least dominant flow), point-to-multipoint or multipoint-to-point (most dominant flow) with over thousands of nodes in the networks.

2.3.1 Design Principles

RPL is useful in LLN application fields because it decouples packet processing and forwarding from its routing optimization objective such as minimization of energy, latency or satisfying constraints. High loss rates, low data rates and instability are examples of constraints associated with LLN. RPL provides a mechanism by which multipoint-to-point traffic from devices within the LLN towards a central control point, in addition to point-to-multipoint traffic from the central point to the devices within the LLN, is supported. Point-to-point traffic, which is the simplest form of traffic, is also supported. The operations of RPL require links that are bidirectional, and a router can be designated as a parent only if its reachability is verified with an external mechanism being activated during the selection process so as to make link properties and neighbor reachability verifiable. Also, it requires that an external mechanism accesses and transports control information (RPL Packet Information) in data packets for enabling the association (mapping) of a data packet with an RPL instance and validation of RPL routing states (e.g. IPv6 Hop-by-Hop RPL Option).

A mechanism that makes dissemination of information over dynamically-formed network topology is created by RPL, thus minimizing configuration in the nodes and making the operation of nodes largely autonomous with the use of ***trickle*** for optimization of the

dissemination process. In addition, RPL assembles topologies of routers that own independent prefixes (in some applications), which may or may not be aggregatable in relation to the routers' origin and the prefix is advertised as on-link. Furthermore, RPL makes it possible to bind a subnet together with a common prefix and to route within the subnet but should not be advertised as on-link because many LLN links have non-transitive properties. With this capability, an authoritative source can infuse information about the subnet into the information to be disseminated by RPL.

Also, IPv6 Neighbor Discovery (ND) information such as Prefix Information Option (PIO) and Route Information Option may be disseminated by RPL. In the dissemination mechanism, the ND information retains all its original semantics for router-to-host, with limited extensions for router-to-router (not routing advertisements), and not to be directly redistributed in another routing protocol. Moreover, host and router behaviors are combined by a RPL node, where it processes the options as a host and advertises the information as a router as needed by the specific link in an ND Routing (Router) Advertisement message.

RPL is independent of any specific features of a particular link layer technology and this satisfies the layered architecture of IP. As a result, it operates on many different link layers, which includes those that are constrained, potentially lossy, or usually used together with highly constrained host or router devices, such as but not limited to, low power wireless or Power Line Communication (PLC) technologies.

2.3.2 Protocol Overview

The RPL protocol uses certain rules to construct its topologies by discovering links and selecting peers since Radio Networks do not actually have predefined topologies. Also, routes in RPL are optimized for traffic to or from one or several roots which act as sinks for the topology. For this reason, the topology is organized or modeled as a Directed Acyclic Graph (DAG) which is divided into one or several Destination Oriented DAGs (DODAGs), with a DODAG per sink. However, if the DAG contains many roots, then the roots are joined or connected by a common backbone like a transit link.

RPL Identifiers

RPL utilizes four variables or values in identifying and maintaining a topology: RPLInstanceID, DODAGID, DODAGVersionNumber and Rank. The RPLInstanceID identifies a set of one or several DODAGs and a particular network may contain many RPLInstanceIDs, with each defining independent set of DODAGs, which could be optimized for different Objective Functions (OFs) and applications. A collection of

DODAGs identified by a particular RPLInstanceID form a RPL Instance, with all its DODAGs using the same OF.

The scope of a DODAGID is also a RPL Instance, with a fusion of RPLInstanceID and DODAGID making a particular DODAG distinct in a network. Moreover, each RPL Instance may contain many DODAGs, with each DODAG having a unique DODAGID.

Furthermore, the scope of a DODAGVersionNumber makes a DODAG, which could be recreated or reconstructed from the DODAG root by means of increasing or incrementing the DODAGVersionNumber. A fusion of DODAGID, DODAGVersionNumber and RPLInstanceID distinctly identifies a DODAG version, which is also the scope of a Rank. The Rank creates a partial order over a DODAG Version by specifying individual node locations in relation to the DODAG root.

Instances, DODAGs and DODAG Versions

As mentioned briefly above, a RPL Instance may have one or more DODAG roots for specific functions or operations. In terms of a single DODAG with one root, the DODAG could be optimized to reduce latency rooted at a centralized entity, e.g., lighting controller in a home automation task or application. In a typical case of multiple uncoordinated DODAGs with independent roots (having unique DODAGIDs), for example, a setup of multiple data collection points in an urban data collection application which do not have appropriate coordinated connectivity with one another or use the combination of many DODAGs by which to dynamically and autonomously partition or divide the network could be achieved or created. Also, with respect to one DODAG with a single virtual root which coordinates LLN sinks (using one DODAGID) over a backbone network, a setup of multiple border routers which work with a reliable transit link to support a 6LowPAN application and being able to behave as logically equivalent interfaces to the sink of that DODAG could be implemented. Every RPL packet is associated with a specific RPLInstanceID and thus a RPL instance.

An example of an RPL Instance which has three DODAGs with DODAG Roots R1, R2, R3 and all advertising the same RPLInstanceID with the lines indicating the connections between parents and children, is shown in Figure 9 below.

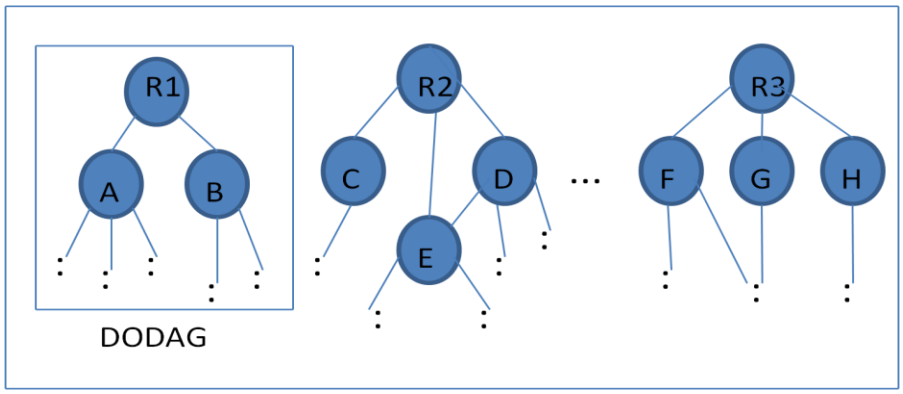


Figure 9: RPL Instance [2]

A DODAG Version is a specific iteration of a DODAG with a given DODAGID. An example of how increasing or incrementing a DODAG Version number results in a new DODAG Version and possibly a different DODAG topology is depicted in Figure 10 below.

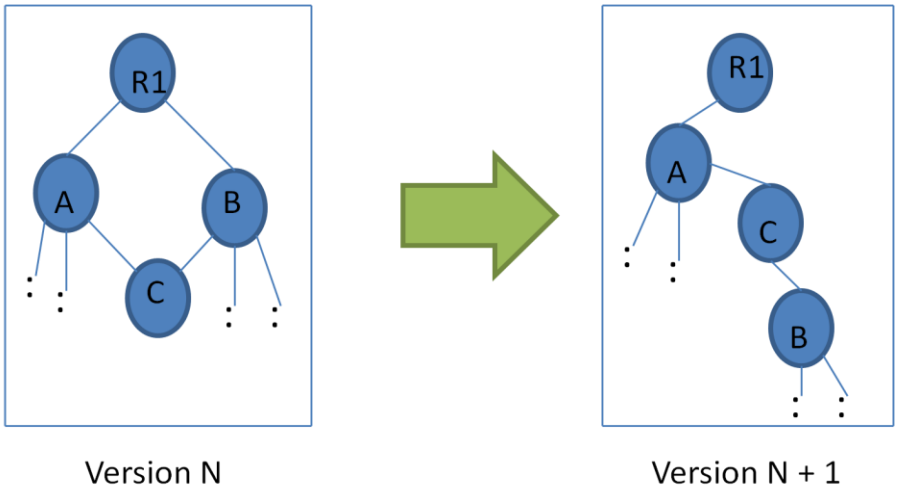


Figure 10: DODAG Version [2]

In Figures 9 and 10 above, the tree-like structure was used in illustrating the concepts so as to simplify them, since a node could have many parents if that connectivity is supported.

2.3.3 Upward Routes and DODAG Construction

RPL routes different information in different directions with respect to DODAG roots and also for DODAG construction. Its provisions are routed up in the direction of DODAG

roots so as to create an optimized DODAG in relation to an OF (Objective Function). Then, DODAG Information Object (DIO) messages are used by RPL nodes in constructing and maintaining the DODAGs.

Objective Function (OF)

The way(s) by which the selection and optimization of routes in a RPL Instance are performed by RPL nodes are prescribed by the OF, which is identified by an Objective Code Point (OCP) in the DIO configuration option. The selection and optimization are done by translating one or more metrics and constraints into a value termed Rank, for approximating a node's distance from one DODAG root.

DODAG Repair

Also, a DODAG root undergoes repairs by initiating the global repair operation with an increment in its DODAG Version Number, which leads to a new DODAG version. As a result, the nodes within the new DODAG Version are able to reposition themselves with a Rank that is not constrained or affected by their previous Rank in the old DODAG Version. In addition, RPL facilitates mechanisms or processes that could be utilized for local repair within a DODAG Version with the use of DIO messages for defining the required parameters or variables as configured from and controlled by policy at the DODAG root.

Security

When it comes to security, RPL facilitates confidentiality and integrity of messages by link-layer mechanisms or uses its own mechanisms in three different modes: unsecured, pre-installed and authenticated security modes. In the "unsecured" mode, RPL sends control message with none of its security mechanism but uses other security mechanisms like the link-layer security to satisfy application-specific security needs. In the "pre-installed" mode, keys that ensure the generation and processing of secured messages are used by nodes that join a RPL Instance. In the "authenticated" mode, pre-installed keys possessed by nodes may only be utilized when a node is joining a RPL Instance as a leaf. Also, a router that joins an authenticated RPL Instance needs to acquire a key from an authentication authority.

Grounded and Floating DODAGs

In RPL, a DODAG could be classified as “grounded” or “floating” depending on the advertisement made by the DODAG root. The grounded DODAG makes connectivity to hosts expected or needed to perform application-specific goal possible whereas the floating DODAG offers routes to nodes in a DODAG in addition to preserving inner connectivity during repair.

Local DODAGs

Optimization of routes to a particular destination in an LLN is achieved by RPL nodes by forming a local DODAG with its DODAG root as the desired destination and the local DAGs having only one DODAG root as compared to global DAGs with multiple DODAG roots. Moreover, the local DODAGs could be constructed on-demand.

Administrative Preference

Depending on the administrative preference, DODAG roots may be deployed in such a way that they could be used over others, which make the facilitation or support of application-specific needs easier.

Datapath Validation and Loop Detection

RPL exhibits on-demand data-path validation and loop detection mechanisms. These mechanisms are on-demand in order to conserve energy since data traffic in LLNs is infrequent. The data packets that contain the RPL Packet Information also include the rank of the transmitter. The loop detection mechanism works in such a way that if there is an inconsistency between the routing decision for a packet (moving up or down) and the rank relationship (value Higher or Lower respectively) existing between the two nodes under scrutiny, then there is a loop. The node that detects the loop then initiates a local repair operation to eliminate the loop.

Distributed Algorithm Operation

RPL uses the Distributed Algorithm Operation to construct the DODAG. In this algorithm, some of the nodes are set as DODAG roots with the appropriate configurations for such a DODAG. Advertisements of their presence, DODAG affiliation, cost of routing, and other metrics are then carried out by the root nodes through **link-local** multicast DIO messages to all the RPL nodes. In accordance with the defined OF and

rank of their neighbors, nodes then join new DODAG so as to select their DODAG parents or maintaining existing DODAG by using the information gathered from the DIOs they listen to. At final stage of the algorithm, through the DODAG parents in the DODAG Version, the nodes create routing table entries for destinations as indicated in the DIO message they capture or receive.

2.3.4 Downward Routes and Destination Advertisement

Downward routes are then created by RPL with Destination Advertisement Object (DAO) messages, which are optional characteristic of P2MP (Point-to-MultiPoint) or P2P (Point-to-Point) traffic applications. Moreover, there are two modes of downward traffic used in RPL: the storing or fully stateful mode and the non-storing or fully source-routed mode. A RPL Instance can be in either mode at a point in time. However, P2P packets move up in the direction of the DODAG Root and then move down in the direction of the final destination in both modes, except if the destination is in the direction of the up route. Furthermore, packets move (up) in the direction of the DODAG root until they reach it before they can move down (towards the destination) in relation to the non-storing mode whereas in the storing mode packets may move in the reverse direction through the influence of the source's common ancestor.

2.3.5 Rank Properties

As mentioned earlier, a node's location in a DODAG Version is represented by a scalar referred to as rank, which helps in avoiding and detecting loops. The computation of the rank, which may depend on parents, link metrics, node metrics, node configuration and policies, is done by the OF (Objective Function) but the rank must implement generic properties despite the OF. In particular, a node's rank must monotonically decrease as the DODAG version is followed in the direction of the DODAG destination, in order for the rank to be seen as the scalar representation of the node's location or radius in the DODAG Version. The rank is an abstract numeric value which represents the node's relative position in the DODAG Version with respect to its neighbors and when the rank is stable as a result of dampening or filtering, then the routing topology is also stable. Furthermore, due to its monotonic increments, the rank could be used the check packet progression from or to the root.

However, in order for a rank to work as expected, it should be compared to other ranks in the DODAG Version, where it is looked at as a fixed point number with the position of the radix point between the integer component and the fractional component being specified by the MinHopRankIncrease. This is the minimum increase in rank between the node and any of its DODAG parents. The DODAG Root creates the MinHopRankIncrease, which in turn provides a tradeoff between the hop cost precision and the maximum number of hops a network could handle. When the MinHopRankIncrease is very big in value, the characterization of a hop's effect on the Rank becomes exact but can only handle a small number of hops.

The rank computation, as stated previously, is the sole responsibility of the OF and it does this by operating on the full 16-bit rank quantity. The calculated rank value is then compared with neighboring rank values to select a parent relationship, detect loops, and track Expected Transmission Count (ETX) with its integer component, which is calculated with the **DAGRank()** macro as shown in the equation below:

$$DAGRank(rank) = \left\lfloor \left(\frac{rank}{MinHopRankIncrease} \right) \right\rfloor,$$

Where *floor(x)* is the function that evaluates to the greatest integer less than or equal to *x*

For example, if a 16-bit rank quantity is decimal 27, and the MinHopRankIncrease is decimal 16, then

DAGRank(27) = $\left\lfloor \left(\frac{27}{16} \right) \right\rfloor = \lfloor (1.6875) \rfloor = 1.$ The integer part of the rank is 1 and the fractional part is $\frac{11}{16}$.

Moreover, with the DAGRank(rank) results, three rank relationships can be established between two neighboring nodes with respect to the DODAG root: less than, equal to and greater than relationships. Given two neighboring nodes A and B, if DAGRank(A) is less than DAGRank(B), then it implies that node A is closer to the DODAG root than node B. Thus, node A may act as a DODAG parent to node B with no loop being created. If DAGRank(A) is equal to DAGRank(B), then both nodes are at similar positions with respect to the DODAG root and this may result in a loop if a particular packet is routed through both nodes. In the third and final instance or relationship, if DAGRank(A) is greater than DAGRank(B), then node B is closer to the DODAG root than node A, and node B can also act as DODAG parent to node A without creating any loop.

2.3.6 Routing Metrics and Constraints used by RPL

When it comes to routing metrics, static and dynamic ones are both needed in LLNs, in addition to link and node metrics. These metrics are utilized by routing protocols in computing shortest paths and also help in the operation of RPL because RPL does not support a single metric or a composite metric which is intended to handle all use cases. Furthermore, RPL makes constrained-based routing possible in cases where constraints are associated with link and the nodes. So in the case whereby a link or node is unable to handle a needed constraint, it is cut-off or "pruned" from the candidate neighbor set and thereby resulting in a constrained shortest path. The OF is responsible for defining the objectives needed to calculate the (constrained) path, in addition to making rules for DODAG parent selection, load balancing and ranking of nodes. Due to the enormous responsibility of the OF, its operation is separated from the routing metrics and the constraints RPL handles. As a result, the collection of metrics and constraints required for determining the appropriate (shortest) path are put in the information the DAG container option in DIO messages convey. Moreover, the resulting preferred path could be a "Shortest path" or a "Shortest Constrained path". The "Shortest path" offers the shortest end-to-end delay whereas the "Shortest Constrained path" offers a (shortest) path which does not traverse any battery-operated node and so optimizes the reliability of that (chosen) path.

2.3.7 Loop Avoidance

When a (RPL) topology undergoes changes, loops may occur but RPL always tries to prevent them during this process and even when they are created, RPL uses rank-based data-path validation mechanisms for their detection. As a consequence, forward progression of packets in the DODAG Version is guaranteed and, repairs are triggered as and when needed.

Nodes may sometimes be greedy by attempting to move deeper (increase their rank) in the DODAG Version so as to have more parents or improvement in some metrics, and may result in instabilities in the DODAG Version. RPL prevents these unwanted conditions by disallowing such misbehaviors once a node joins the DODAG Version.

For example, a node may be willing to receive and process a DIO message from a node in its own sub-DODAG and in general a node deeper than itself. In such a scenario, a possibility exists that a feedback loop could be created, wherein two or more nodes continue to try and move in the DODAG Version while attempting to optimize against each other. This could lead to instability in the network. It is for this reason that RPL puts a limit on the cases where a node processes DIO messages from nodes that are deeper in the DODAG Version to some forms of local repairs in a condition known as "event horizon". The "event horizon" is the situation in which the node is unable to be influenced

beyond the set limit that could result in instability through the action of nodes that may reside within its sub-DODAG.

Greedy DODAG parent selection which could also result in instability is illustrated in Figure 11 below.

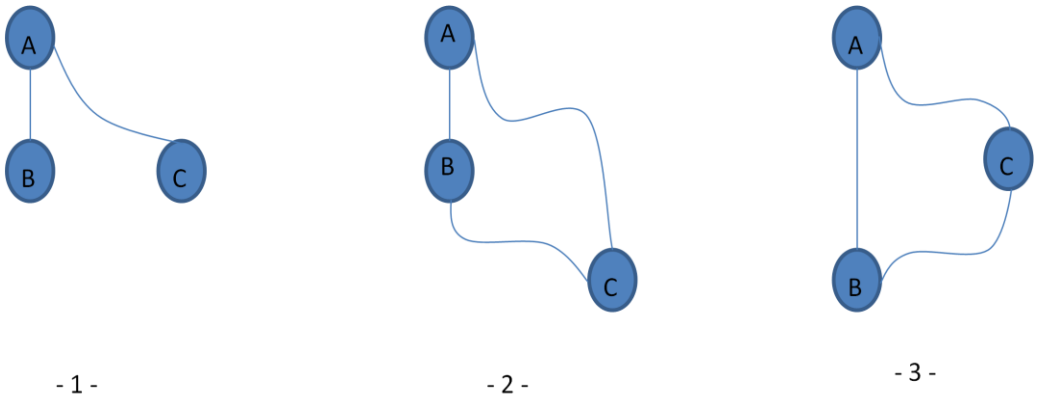


Figure 11: Greedy DODAG Parent Selection [2]

Figure 11 depicts a DODAG in 3 different configurations. A usable link between (B) and (C) exists in all 3 configurations. In Figure 11-1, Node (A) is a DODAG parent for Nodes (B) and (C). In Figure 11-2, Node (A) is a DODAG parent for Nodes (B) and (C), and Node (B) is also a DODAG parent for Node (C). In Figure 11-3, Node (A) is a DODAG parent for Nodes (B) and (C), and Node (C) is also a DODAG parent for Node (B).

Using Figure 11 as an example to illustrate the greedy condition intending to optimize for 2 parents, with Figure 11-1 being the initial condition:

> *Suppose Node (C) first is able to leave the DODAG and rejoin at a lower rank, taking both Nodes (A) and (B) as DODAG parents as depicted in Figure 11-2. Now Node (C) is deeper than both Nodes (A) and (B), and Node (C) is satisfied to have 2 DODAG parents.*
>
> *Suppose Node (B), in its greediness, is willing to receive and process a DIO message from Node (C) (against the rules of RPL), and then Node (B) leaves the DODAG and rejoins at a lower rank, taking both Nodes (A) and (C) as DODAG parents. Now Node (B) is deeper than both Nodes (A) and (C) and is satisfied with 2 DAG parents.*
>
> *Then Node (C), because it is also greedy, will leave and rejoin deeper, to again get 2 parents and have a lower rank than both of them.*

Next Node (B) will again leave and rejoin deeper, to again get 2 parents

And again Node (C) leaves and rejoins deeper …

The process will repeat, and the DODAG will oscillate between Figure 11-2 and Figure 11-3 until the nodes count to infinity and restart the cycle again.

This cycle can be averted through mechanisms in RPL:

Nodes (B) and (C) stay at a rank sufficient to attach to their most preferred parent (A) and don't go for any deeper (worse) alternate parents (Nodes are not greedy)

Nodes (B) and (C) do not process DIO messages from nodes deeper than themselves (because such nodes are possibly in their own sub-DODAGs).

In RPL, DODAG loops may also be present if nodes detach from DODAGs and re-attach to devices in their prior sub-DODAGs, as well as if DIO messages are missed or if local repair mechanisms are being used. However, a DODAG loop can be eliminated through strict adherence to the DODAG Version Number.

Finally, DAO loops could also occur if a parent creates a route after the reception and processing of a DAO message from a child, but that child had already removed the associated DAO state and thus a No-Path being lost and remains until all states have been removed or cleared. RPL, however, adds an optional mechanism for DAO messages acknowledgement to alleviate the effect of a lost DAO message. Also, RPL adds loop detection mechanisms to ease the effect of DAO loops and as well start fixing or removing the loops when necessary. [10]

2.4 Summary

In the web, such as HTTP, there is plethora of applications that could be implemented. However, in a restricted environment, such as LLN, the limited resources must be efficiently utilized to ensure the proper and continuous functioning of the network. Hence, the use of light-weight protocols such as CoAP (constrained applications) and RPL (routing) in such networks, based on the REST architecture.

As these protocols are designed to support an "Internet of Things", we will investigate their suitability in practice in the remainder of this thesis and compare different modes of operations.

3. Research Overview

This chapter gives an overview of the research carried out with respect to the experiments performed, the evaluation methodology used, and the building and use of a testbed.

3.1 Experiments Performed

The operating system used for this study is the Contiki OS. The Contiki OS is the open source operating system for the "Internet of Things". It runs on networked embedded systems and WSNs. This OS (Operating System) provides IP communication, both for IPv4 and IPv6, and its uIPv6 stack (fully tested) is IPv6 Ready Phase 1 certified. Furthermore, its power-efficient radio mechanisms such as ContikiMAC, make it possible for battery-operated devices to participate in IPv6 networking. Also, Contiki supports 6lowpan header compression, IETF RPL IPv6 routing, IETF CoAP application layer protocol, and many other protocols. It is written in the C programming language and has an event-driven kernel, on top of which application programs can be dynamically loaded and unloaded at run time. Its processes are lightweight and so provide a linear, threadlike programming style on top of the event-driven kernel. In addition, it supports per-process optional multithreading and inter-process communication using message parsing. Contiki also provides a regular "malloc()" operation, memory block allocation and a managed memory allocator for memory management. [3]

The device used for building the wireless sensor network was the Redbee-Econotag and below is a picture of it.

It is a Freescale MC13224v ARM7 microcontroller with, among others, 802.15.4 radio, integrated bootloader, PCB antenna (open-air line-of-sight range approximately 500 ft at 0 dBm; full transmit power is 4.5 dBm), 24 MHz crystal, 2 Light-Emitting-Diodes for general purpose or for receiving and transmitting data, and flash erase jumpers. [4]

The experiments were carried out to investigate the differences between the PULL and PUSH protocols in a constrained or restricted environment, and to select the better protocol, if

applicable. The protocols to be studied are RPL (IPv6 Routing protocol for low power and Lossy Networks – for routing), CoAP (Constrained Application Protocol - for the PULL mechanism) and the usual uIPv6 UDP (for the PUSH mechanism). In the experiments, ten RedBee-Econotags (Freescale MC13224v ARM7 microcontroller with 802.15.4 radio) were organized or setup in six different topologies, loaded with a testbed and nine different traffic rates applied to each topology. The six topologies were selected or chosen because they were the most reasonable and appropriate setups for the limited number of nodes available. Furthermore, the chosen topologies could provide initial cues for the relative behavior of the two protocols and for the initial bits scale (especially in cases where things go wrong). These experiments were the same for both protocols and the results or data collected saved to a file with timestamps. Detailed descriptions of all these can be found in the next chapter.

3.2 Evaluation Methodology

The results gathered were then evaluated by comparing the latency of the data collected during the least traffic rate to the remaining eight traffic rates. This was to show the effect of increasing traffic rate on latency. Also, the stability of the network was observed as the traffic rate was incremented. These were done for both protocols.

3.3 Building and Using the Testbed

The testbed comprises of the server side and the client side. For the server side, a UDP server and a REST server were combined into one system and configured over-the-air. The over-the-air configuration was used only by the PUSH protocol, since the PULL protocol did not need any configuration due to its request-reply model. In addition, both protocols had the same size of data to send and it was 12 bytes in all cases. On the other hand, the clients were separated due to the differences in their mode of communication. Also, the UDP client was what was used to send the configuration parameters and control messages for stopping and restarting the tests. Detailed descriptions of all these can be found in chapters 4 and 5.

4. The Testbed

This chapter describes the components of the testbed used for the tests. The experiments were carried out to investigate the differences between the PULL and PUSH protocols in a constrained or restricted environment, and to select the better protocol, if applicable. In the experiments, ten RedBee-Econotags (Freescale MC13224v ARM7 microcontroller with 802.15.4 radio) were organized or setup in six different topologies, loaded with a testbed and nine different traffic rates applied to each topology. The testbed comprises of a combined normal UDP server and REST server with automated clients. In this system, the UDP client is used for sending the configuration parameters (string) to the combined UDP and REST servers but only used by the UDP server (PUSH protocol), since CoAP does not need any configuration parameters because it is a PULL protocol. Figure 12 below is a picture of the whole system.

For the UDP client-server communication, the client first sends the configuration parameter to the server and then a user-button on the client device is pressed to start the simulation. In addition, the user-button can also be used to stop and restart the tests. The server then sends the results or test data to the client for storage in a file, with timestamps.

For the CoAP client-to-REST server communication, the client sends resource requests to the server at specified times, equal to that of the UDP, and stores the received replies in a file with timestamps. In addition, both protocols had the same size of data to send and it was 12 bytes in all cases.

The results gathered were then evaluated by comparing the latency of the data collected during the least traffic rate to the remaining eight traffic rates. This was to show the effect of increasing traffic rate on latency. Also, the stability of the network was observed as the traffic rate was incremented.

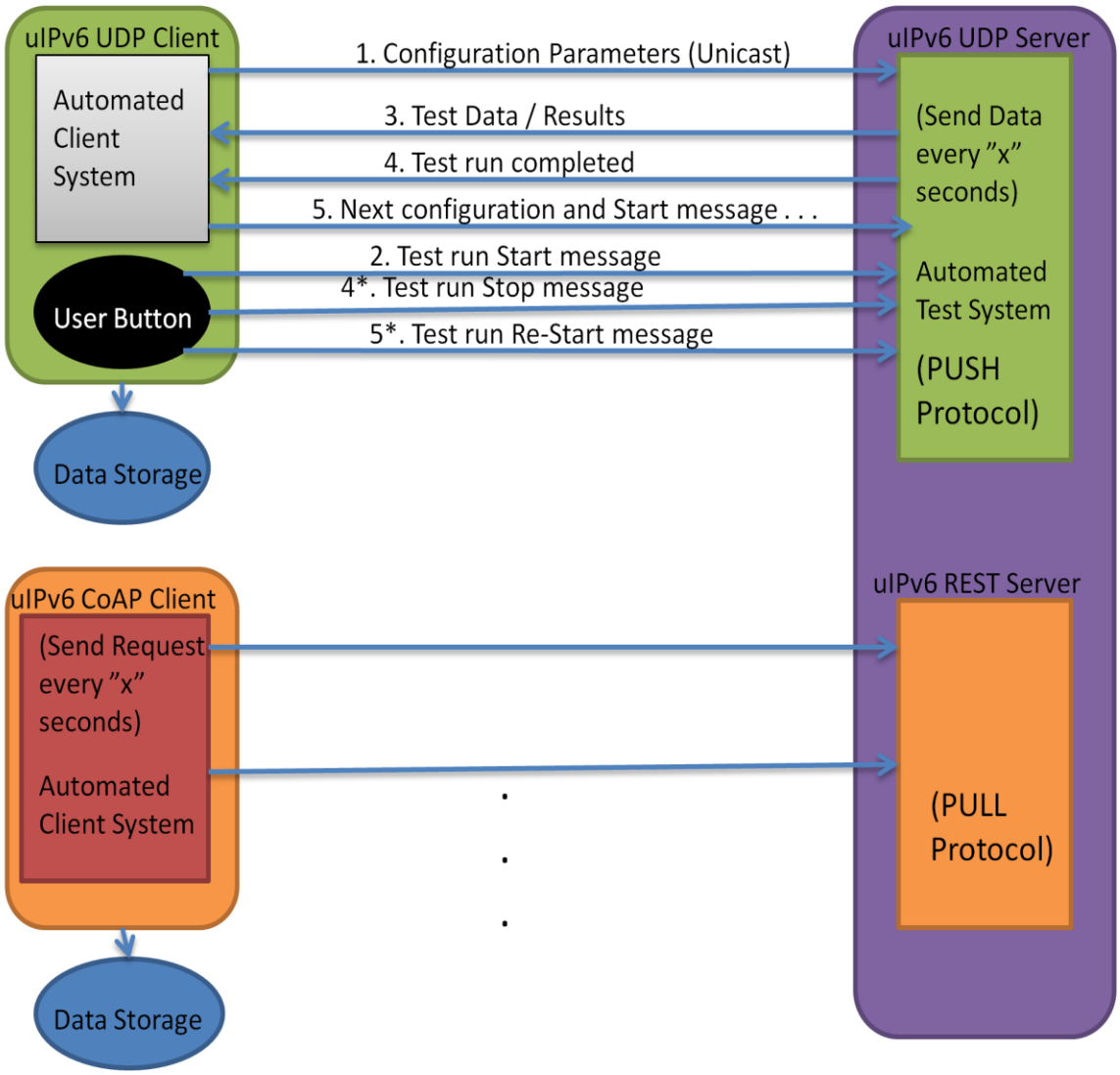


Figure 12: The Test-Bed

4.1 The UDP Client (Sink)

This client automatically sends the configuration parameters (string) to the server ten seconds after being powered on, using a timer. Below is how the parameters (string) look like.

```
"Cnf, 360, Pw, Pi, 2, Si, 1, Tm, Ti, 4, Si, 1, Lg, Li, 6, Si, 1, %d"
```

Below is what the symbols and numbers above signify.

Symbol	Meaning
Cnf	Configuration message
360	The number of times this configuration should run (periodic)
Pw	Power
Pi	instantaneous Power
2,4,6	The timer which indicates when the instantaneous quantity should be read and it is in seconds. For example, 2,4,6 timers mean read instantaneous Power after 2 seconds and stop the timer, read instantaneous Temperature after 4 seconds and stop the timer, read instantaneous Light after 6 seconds and stop the timer, and then restart all the timers to repeat the sequence. Thus, reading and sending a value every 2 seconds.
Si	Send instantaneous value
1	The specific period in which the value should be sent (1 in this case means the value should be sent in every period or run)
Tm	Temperature
Ti	instantaneous Temperature
Lg	Light
Li	instantaneous Light
%d	The number of different configuration parameters programmed on the client for automated sending after an above configuration run has been completed and a "Done" message received from the server

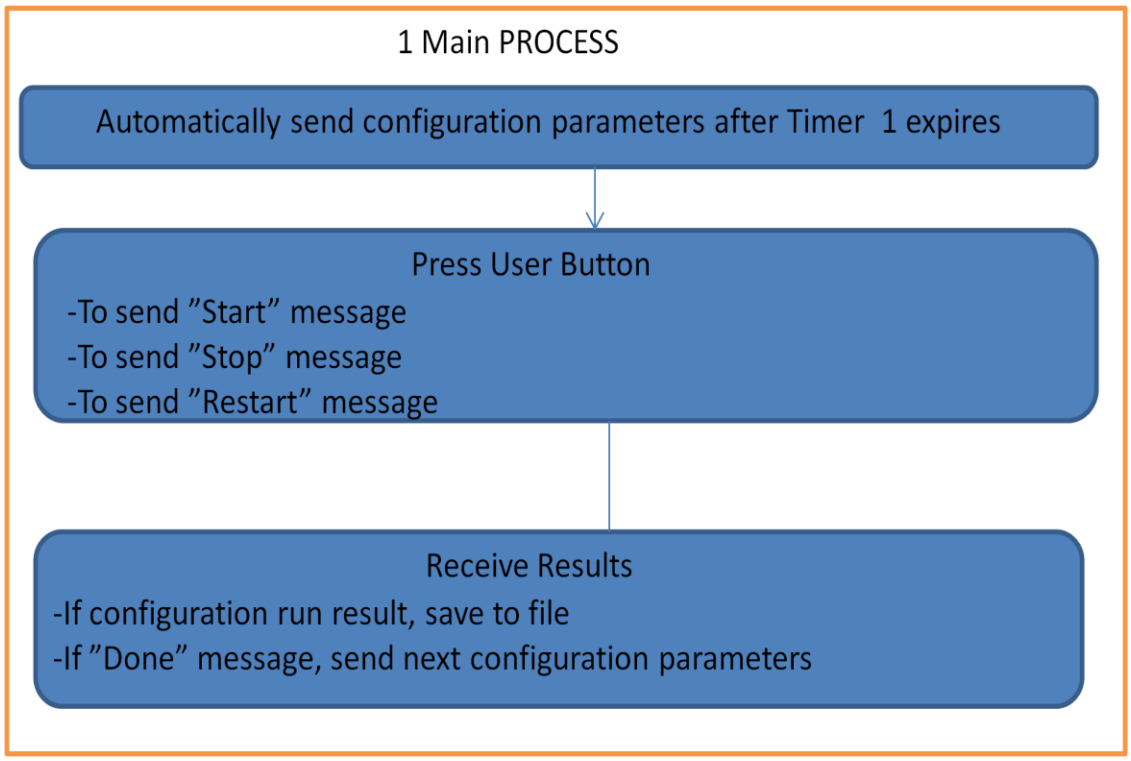
Furthermore, to start the simulation a "Start" message is sent by pressing a user-button on the client node and below is the string.

```
"Start, 360, Pw, Pi, 2, Si, 1, Tm, Ti, 4, Si, 1, Lg, Li, 6, Si, 1, %d"
```

As seen in the string above, it is the same as the previous one with the only difference being the first field "Cnf" which is replaced with "Start". This makes the automatic sending of new configuration parameters, after an above configuration run has been completed and a "Done" message received, simpler since there is no use in sending configuration and start messages separately automatically. Thus, a reduction in the number of sent messages.

In addition, the user-button can also be pressed to stop the simulation and pressed again to restart the simulation. The results received from the simulation are then saved to file with timestamps.

Below is the structure of the client code.



The complete source code is included as “Appendix 1”.

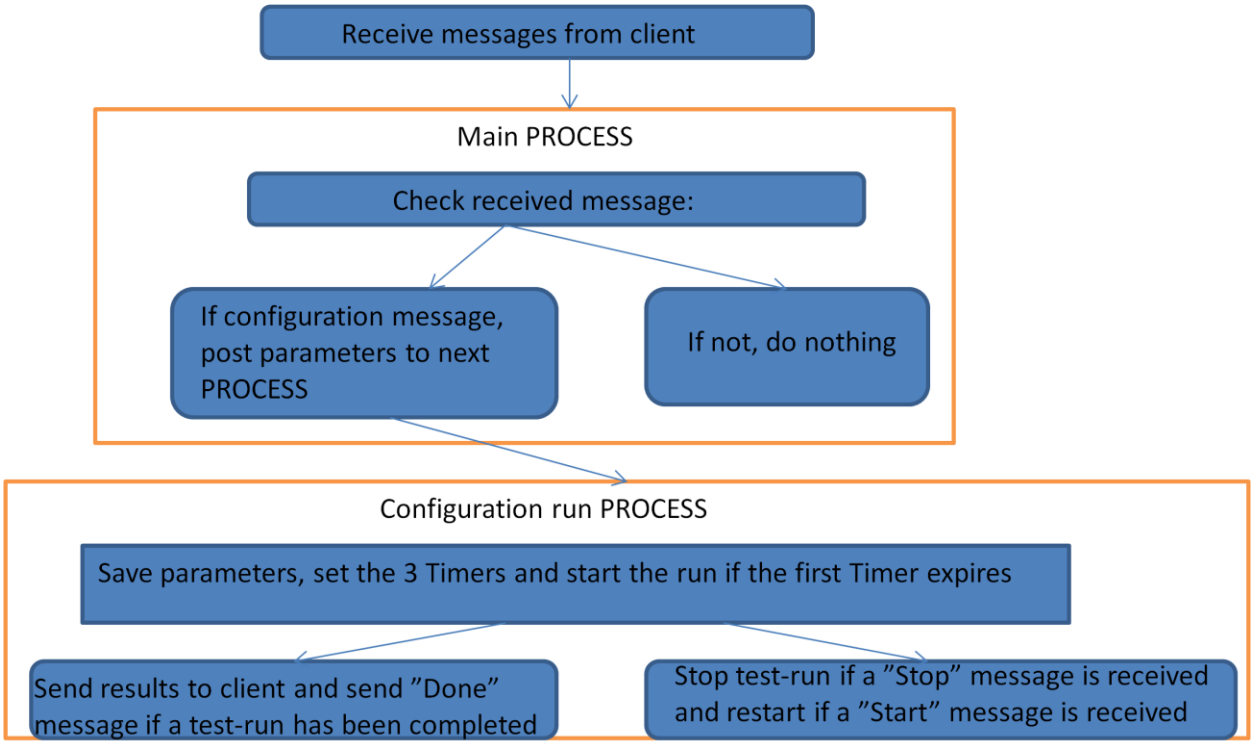
However, this code only worked for one-client-to-one-server scenario because uIPv6 multicast applications are not yet supported (at least on Contiki OS). So a modified version of a “Sender” and a “Sink” source codes which come with the Contiki OS were used for the many-senders-to-one-sink scenarios (hard-coded).

4.2 The UDP Server

The server, upon receipt of the configuration parameters (string), uses the “,” (comma) separator to place each field in the string into a separate character-string buffer for use by their specified operators. It then sets the appropriate timers and starts the test-run when a “Start” message is received from the client node.

Moreover, the total number of different tasks to be selected for a configuration run was 9 (instantaneous Power, average Power, total Power, instantaneous Temperature, average Temperature, total Temperature, instantaneous Light, average Light, total Light) but was reduced to only 3 due to the size of code, so as to make it fit into memory. The results are then sent after every “x” seconds as specified in the configuration parameters.

Also, a “Done” message is sent to the client node anytime a configuration run has been completed and if a new configuration string is expected. Below is the structure of the server code.

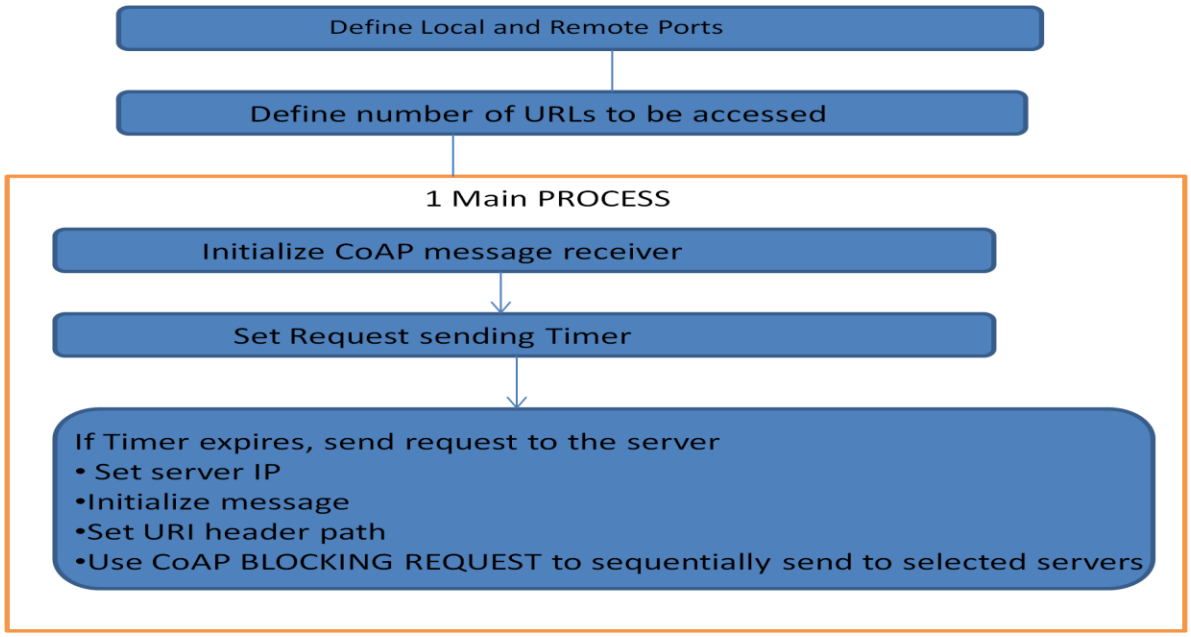


The complete source code is included as “Appendix 2”.

4.3 The CoAP Client

This client just requests resources from the server after a specified time (periodically) and the source code used was a slightly modified version of the CoAP client which comes with the Contiki OS. The code was used for the one-client-one-server and many-clients-to-one-server scenarios.

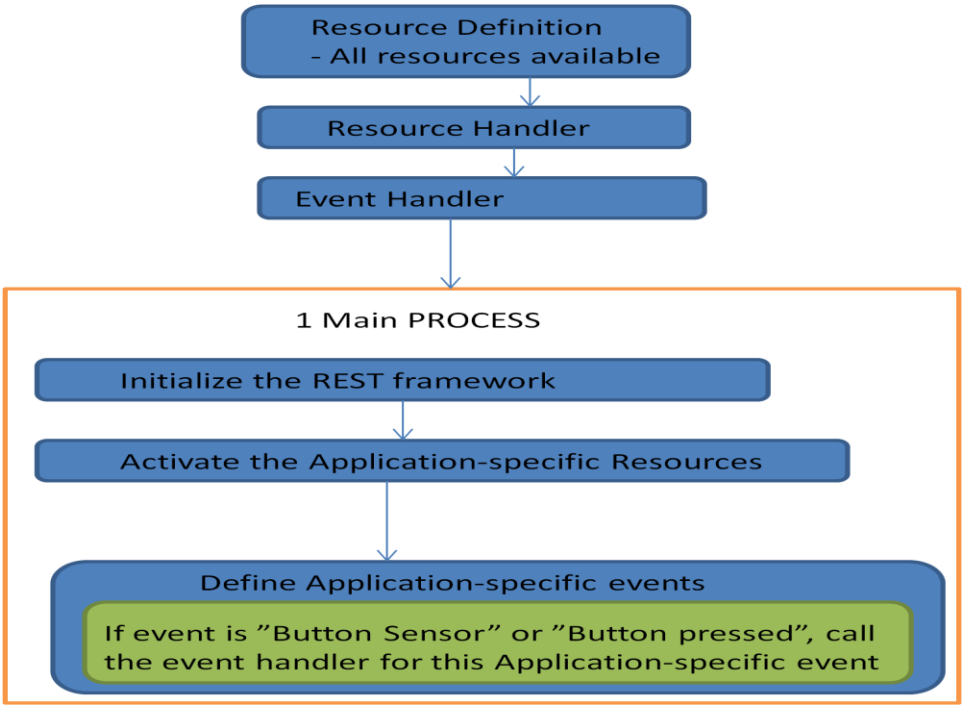
In the case of one-client-to-many-servers, the code was modified a little bit more as compared to the first part. An array of server IP addresses was first created for 9 different servers, since there were only 10 nodes and one used as the client. Then, the addresses were specified and CoAP-Blocking-Request was then used to send the requests to the servers in sequence with the help of an “if” statement and an integer buffer. Below is the structure of the client code.



The complete source code is included as “Appendix 3”.

4.4 The REST Server

The server receives the request from the client, checks the requested resource URI and if found, sends appropriate response to the client. Below is the structure of the server code.



The code used was a slightly modified version of the REST server that comes with the Contiki OS.

5. Test Cases (Setups)

This chapter shows the different test cases used in this thesis. The comparative test cases were six in number (because these were the most reasonable setups that could be constructed from them due to limited number of nodes available), in which different setups were investigated to see how the two different protocols, normal udp client-server (sender-sink) and CoAP client-server (CoAP client – REST server), behave with respect to the rate (frequency) of traffic within different network topologies. The rate (frequency) of traffic was from 10 seconds to 5 milliseconds and each run lasted for an hour. Also, each traffic rate for a test case was repeated three times. The results will be presented in the next chapter. Below is an overview table of the components involved in each test case.

Use Cases	**Number of Senders or Clients**	**Number of Sinks or Servers**	**Number of RPL Routers**	**Traffic Rates**
Case 1	1	1	2	10, 5, 3, 2 and 1 (seconds), 50, 20, 10 and 5 (milliseconds)
Case 2	1	1	8	Same as previous
Case 3	3	1	6	10, 5, 3, 2 and 1 (seconds), 5 (milliseconds)
Case 4	3	3	4	Same as previous
Case 5	1	9	-	10, 5, 3, 2 and 1 (seconds), 50, 20, 10 and 5 (milliseconds)
Case 6	9	1	-	1 second

Below are the diagrams of the test/use cases, starting from the simplest.

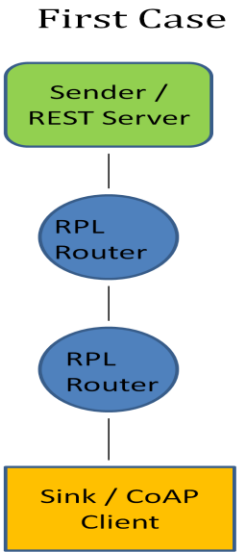


Diagram 1.Use Case 1: One Client to One Server with Two RPL Routers

Use Case 1, as depicted in Diagram 1 above, was the simplest network topology or test case in the study of the two protocols and this setup was chosen because the effect of multi-hopping on latency was the most important part of this research. The setup comprises of a UDP sender / REST server to a Sink / CoAP client respectively, with two RPL routers. Nine different rates (frequencies) of traffic of 10 seconds, 5 seconds, 3 seconds, 2 seconds, 1 second, 50 milliseconds, 20 milliseconds, 10 milliseconds and 5 milliseconds were the runs for this setup. An hour was the duration for times 10 seconds to 1 second whereas a minute was the duration for times 50 milliseconds to 5 milliseconds. Table 1 further below contains the results.

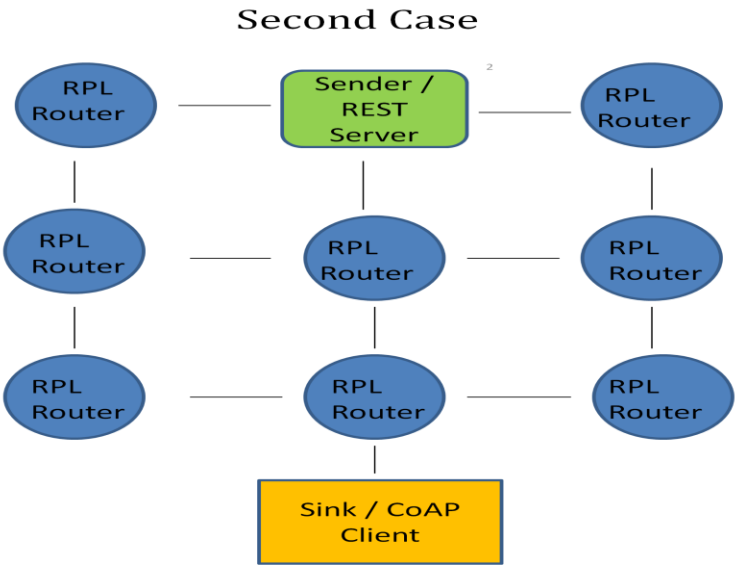


Diagram 2.Use Case 2: One Client to One Server with Eight RPL Routers

Use Case 2, as shown in Diagram 2 above, comprises of a UDP sender / REST server to a Sink / CoAP client respectively, with eight RPL routers. This setup was selected because the effect of multi-hopping and multi-routing on latency were important part of this research. The same traffic rates and time durations for Diagram 1 were also used in this scenario, with the results tabulated in Table 2.

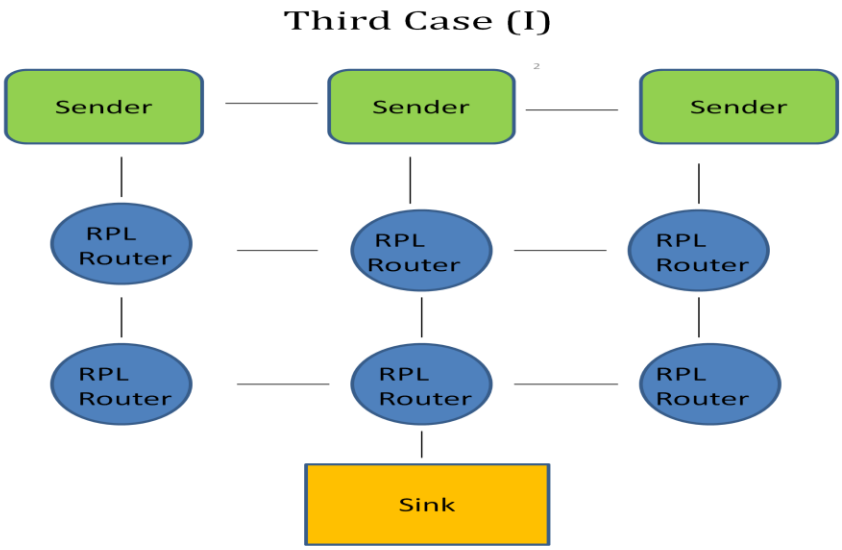


Diagram 3.Use Case 3(I): One Sink to Three Senders with Six RPL Routers

Use Case 3(I), as drawn in Diagram 3 above, is made up of three UDP senders and a Sink with six RPL routers. This setup was selected because the effect of multi-hopping and multi-routing on latency were important part of this research. Also, this test case had six different traffic frequencies of 10 seconds, 5 seconds, 3 seconds, 2 seconds, 1 second and 5 milliseconds with their corresponding time durations as stated above for Diagrams 1 and 2. The results for this part are shown in Table 3.

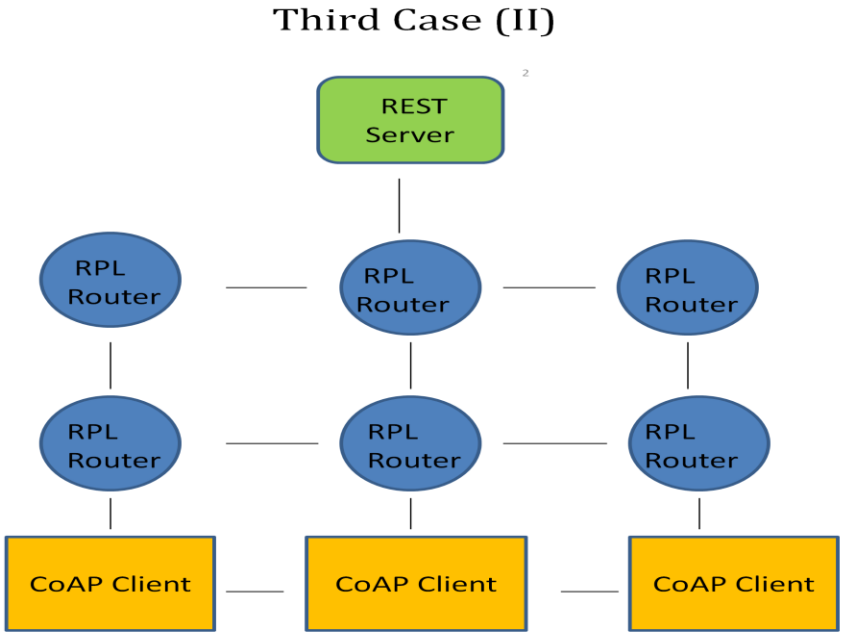


Diagram 4.Use Case 3(II): Three Clients to One Server with Six RPL Routers

Use Case 3(II), as pictured in Diagram 4 above, is a network of three CoAP clients and a REST server with six RPL routers. This setup was selected because the effect of multi-hopping and multi-routing on latency were important part of this research. Also, this PULL protocol setup is equivalent to the PUSH protocol setup in Diagram 3 further above and so everything related to Diagram 3 applies to this scenario.

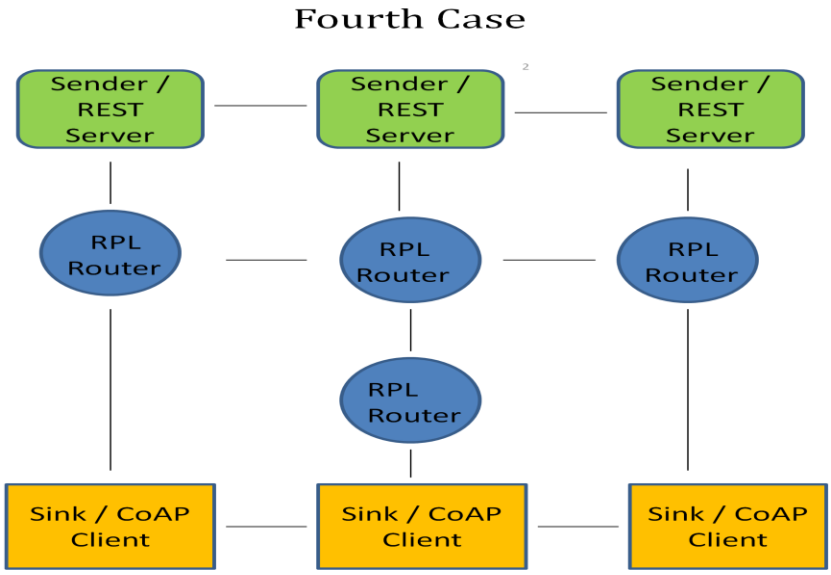


Diagram 5.Use Case 4: Three Sinks/Clients to Three Senders/Servers with Four RPL Routers

Use Case 4, as visualized in Diagram 5 above, contains three Sinks to three Senders or three CoAP clients to three REST servers for the PUSH and PULL protocols respectively, with four RPL routers. This setup was selected because the effect of multi-hopping and multi-routing on latency were important part of this research. The traffic rates and time durations used were the same as those in Diagram 4 above, with the results recorded in Table 4.

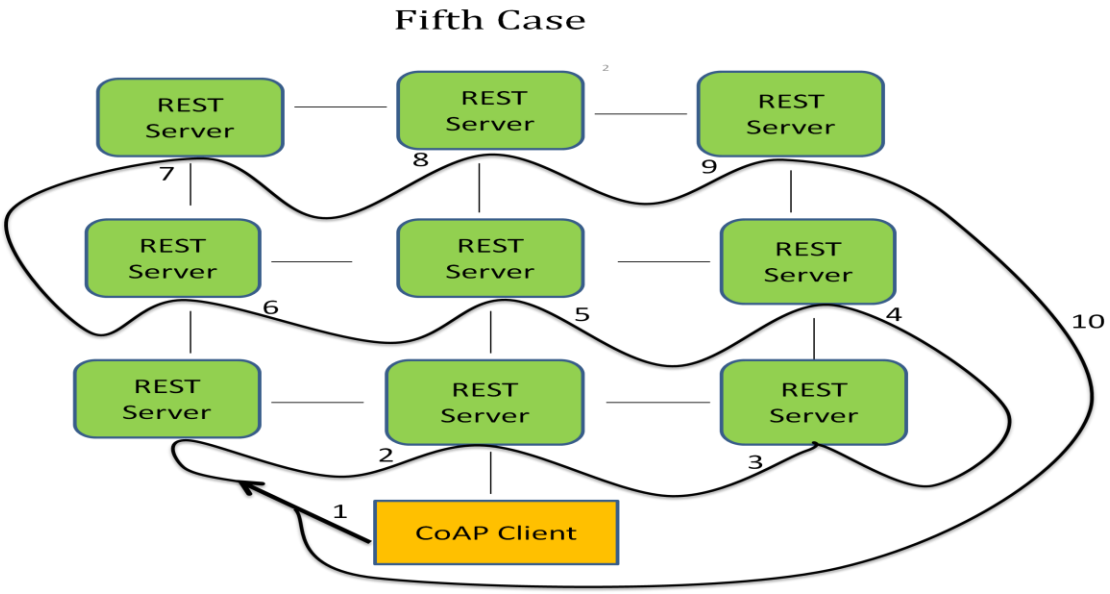


Diagram 6.Use Case 5: One Client to Nine Servers

Use Case 5, as depicted in Diagram 6 above, comprises of one CoAP client and nine REST servers. This setup was selected because the effect of multi-hopping and multi-routing on latency were important part of this research. The rates of traffic and time durations for this test case were the same as used in Diagram 2, and the results written into Table 5.

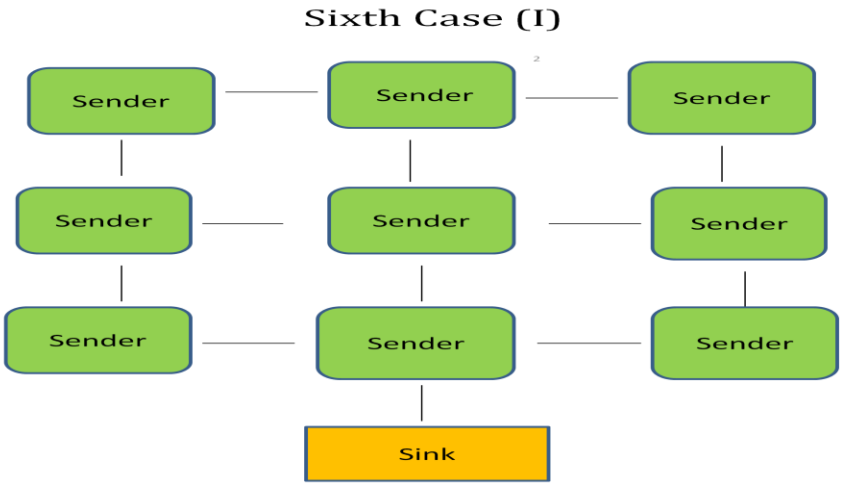


Diagram 7.Use Case 6(I): Nine Senders to One Sink

This PUSH protocol setup in Diagram 7 above has nine UDP senders and a SINK with a traffic rate of 1 second and time duration of an hour. This setup was selected because the effect of multi-hopping and multi-routing on latency were important part of this research. Table 6 contains the result for this test case.

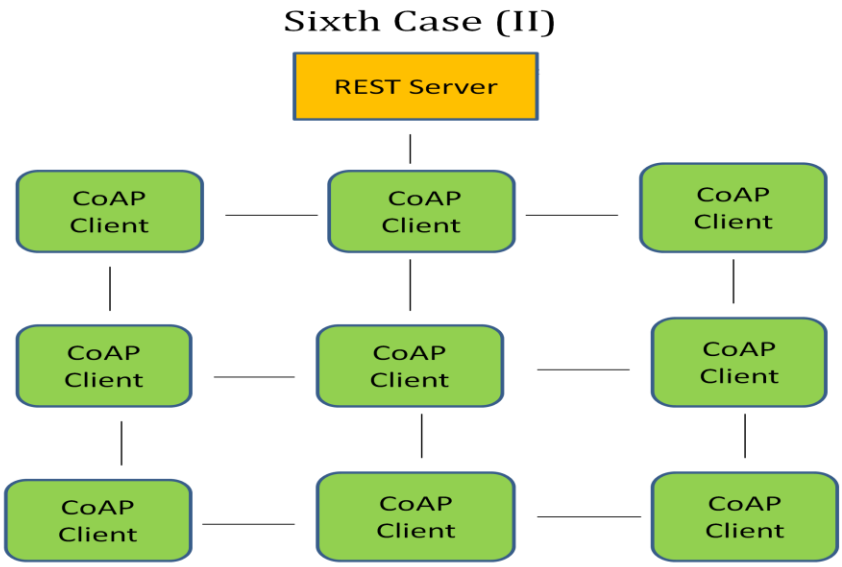


Diagram 8.Use Case 6(II): Nine Clients to One Server

The PULL protocol setup in Diagram 8 above is equivalent to the PUSH protocol setup in Diagram 7 further above and so everything related to Diagram 7 applies to Diagram 8. This setup was selected because the effect of multi-hopping and multi-routing on latency were important part of this research.

6. Tables of Results

This chapter presents the results of the test cases outlined in the previous chapter, starting from the first use case.

Rate	**10s**	**5s**	**3s**	**2s**	**1s**	**50ms**	**20ms**	**10ms**	**5ms**
CoAP (RTT delay / ms)	6	6	6	6	6	6	6	6	8
UDP (delay / ms)	1	1	1	1	1	1	1	1	1

Table 1.Use Case 1: One Client to One Server with Two RPL Routers

From Table 1 above, it could be seen that the PUSH protocol (UDP) has a delay of 1 ms whereas the PULL protocol (CoAP) has an RTT delay of 6 ms when the traffic rate is greater than 6 ms and 8 ms when traffic rate is a millisecond less than its usual RTT. This means that the PUSH protocol, in such a setup, can accommodate a traffic rate of 1 ms without any problem and as a result the sink can receive a total of 1000 data points per second. On the other hand, the PULL protocol, in such a setup, can also handle a maximum traffic rate of 6 ms without any effect on its usual RTT and a total of 166 data points per second could be gathered. However, in the case of the PULL protocol, a traffic rate of 5 ms would result in a total of 125 data points per second (41 data points lesser than the usual case).

Rate	**10s**	**5s**	**3s**	**2s**	**1s**	**50ms**	**20ms**	**10ms**	**5ms**
CoAP (RTT delay / ms)	6	6	6	6	6	6	6	6	8
UDP (delay / ms)	1	1	1	1	1	1	1	1	1

Table 2.Use Case 2: One Client to One Server with Eight RPL Routers

Also, the same effect as seen and explained in Table 1 further above applies to Table 2 above because the two scenarios are similar with respect to the source and destination (one-to-one mapping).

Rate	**10s**	**5s**	**3s**	**2s**	**1s**	**5ms**
CoAP (RTT delay / ms)	6	6	6	6	6	(8 – 9)*
UDP (delay / ms)	1	1	1	1	1	1*

Table 3.Use Case 3: Three Senders to One Sink with Six RPL Routers and Three Clients to One Server with Six RPL Routers.

Table 3 above has the same results as in Tables 1 and 2 for traffic rates from 10 seconds to 1 second. However, for a rate of 5ms, only the UDP sender which started sending first worked

and also the CoAP client that started requesting for resources first worked. Furthermore, when that UDP sender was stopped another sender then took over the sending and the same applied to the CoAP client. The reason for this behavior may be that the 5 ms traffic rate is really fast and so the other nodes did not get the chance to really initiate any operation once a node is in operation (sending or requesting) or their data just got lost. Due to this effect, the PULL protocol had RTT delay of 8 to 9 milliseconds whereas the PUSH protocol had its usual 1 millisecond delay.

Rate	**10s**	**5s**	**3s**	**2s**	**1s**	**5ms**
CoAP (RTT delay / ms)	6	6	6	6	6	8*
UDP (delay / ms)	1	1	1	1	1	--

Table 4.Use Case 4: Three Sinks/Clients to Three Senders/Servers with Four RPL Routers

Also, Table 4 above has the same results as in Tables 1, 2 and 3 above for traffic rates from 10 seconds to 1 second. However, for a rate of 1 second, only one of the UDP Senders worked at first when the three sending nodes were started but when they were all stopped and restarted, they all started working as expected. This test was done several times, as done for all, but the same behavior recurred. Also, for a rate of 5ms, none of the UDP Senders worked but only the CoAP client that started requesting for resources first worked at all times with an RTT delay of 8 milliseconds. The results were surprising since all the sources (senders and clients) had dedicated destinations (sinks and servers) and so should not have had any conflict. Maybe, this behavior was due to network instability as a result of the high traffic rate (5ms). In addition, in the case of the PULL protocol, it seems the client-server pair that starts interacting before the rest takes control of the network and thereby prevents the others from communicating.

Rate	**10s**	**5s**	**3s**	**2s**	**1s**	**50ms**	**20ms**	**10ms**	**5ms**
CoAP (RTT delay / ms)	6	6	6	6	6	6	6	6	8
UDP (delay / ms)	1	1	1	1	1	1	1	1	1

Table 5.Use Case 5: One Client to Nine Servers (CoAP) and Use Case 2: One Client to One Server with Eight RPL Routers (UDP)

Table 5 above also had the same results as recorded in Table 2 further above. This is expected because both scenarios are similar in their operation

Rate	**1s**
CoAP (RTT delay / ms)	6 - 7
UDP (delay / ms)	1 - 2

Table 6.Use Case 6: Nine Senders to One Sink (UDP) and Nine Clients to One Server(CoAP)

Finally, Table 6 above resulted in RTT delay of 6 to 7 milliseconds for the PULL protocol whereas the PUSH protocol had delay of 1 to 2 milliseconds. A reason which could be assigned to this behavior is that the increased number of nodes (senders or clients up to 9) created the extra 1 millisecond delay in both protocols. The 1 second traffic rate was the only rate used in this scenario because the lower rates (10, 5, 3 and 2 seconds) would have resulted in a normal delay as seen in the others and the higher rates (50, 20, 10 and 5 milliseconds) would have resulted in the behavior experienced in test case 3.

7. Discussions

As seen in Diagrams 1 to 8 further above, the maximum number of nodes used for the tests was ten and this was due to monetary, distance (shipment) and customs clearance constraints. However, one does not actually need a large number of nodes to run these tests because depending on the type of tests and how well they are executed, it could provide initial cues for the relative behavior of the two protocols and for the initial bits scale (especially in cases where things go wrong).

Also, the static routing tables were tested with a source code provided by one of the developers of the Contiki OS (which comes with the OS) and so the network topologies (multi-hop routing) are just as shown in Diagrams 1 to 8.

From the results and analysis, it was very clear that the PULL protocol performed normally in most of the cases and also had better performance than the PUSH protocol even in cases where the network was loaded with enormous amount of traffic in a short space of time.

Furthermore, the PULL protocol having an RTT of 6 milliseconds in normal cases as compared to a delay of 1 millisecond for the PUSH protocol is very reasonable because of its request-reply communication model.

In addition, in terms of data gathering, the PULL protocol would be much easier to manage than the PUSH protocol because any possible change in the rate of data gathering can be done on the client side (which is closer to the user) as compared to the latter where changes would have to be done on all the senders.

Finally, in terms of efficient energy usage, the PULL protocol is likely to consume less energy than the PUSH protocol because its servers would most of the time be “resting” as compared to the latter where the senders need to permanently be attentive and wait for incoming messages.

8. Conclusion

It was very clear that the PULL protocol performed normally in most of the cases and also had better performance than the PUSH protocol even in cases where the network was loaded with enormous amount of traffic in a short space of time.

In addition, the PULL protocol having an RTT of 6 milliseconds in normal cases as compared to a delay of 1 millisecond for the PUSH protocol is very reasonable because of its request-reply communication model.

However, in a network of many clients/senders to many servers/sinks respectively, both protocols performed poorly when the traffic rate was very high (5 ms), though the performance of the PULL protocol was better than that of the PUSH protocol. This could mean that, in such a network (which is usually the case in real wireless sensor networks), the traffic rate should not be too high but about a maximum of 1 second. Also, the abnormal behavior of the two protocols in this situation could mean that these protocols may not be able to withstand very heavy traffic rates or may not function well in very busy wireless sensor networks.

So, with the aforementioned observations, it seems the PULL protocol (CoAP) performed better than the PUSH protocol (normal UDP). Moreover, future wireless sensor networks are more likely to use CoAP in their operations.

In the end, the purpose of this work, which was to compare the PULL and PUSH protocols used in wireless sensor networks and select the better protocol, was achieved or accomplished.

The total number of nodes used was just 10 and so it would be difficult to generalize the results of this research. Therefore, to have results which could be applied to real wireless sensor networks, a total of about 100 nodes should be used. Also, since uIPv6 multicast applications are not yet supported (at least on Contiki OS), an implementation of support for multicast applications would help a great deal in understanding the behavior of multicasting in the “Internet of Things”, using the aforementioned technologies and/or protocols.

References

[1]Geoffrey Werner- Allen, Patrick Swieskowski, Matt Welsh (2005). Motelab: A Wireless Sensor Testbed. *IPSN '05 Proceedings of the 4th International Symposium on Information, Article No. 68* http://dl.acm.org/citation.cfm?id=1147685.1147769 on 20.02.2012.

[2] Jennifer Yick, Biswanath Mukherjee, Dipak Ghosal (2008). Wireless Sensor Network Survey. *Department of Computer Science, University of Carlifornia, Davis, CA 95616, United States* http://www.sciencedirect.com/science/article/pii/S1389128608001254 on 21.02.2012

[3] Adam Dunkels, Oliver Schmidt, Niclas Finne, Joakim Eriksson, Fredrik Österlind, Nicolas Tsiftes, Mathilde Durvy (2012). The Contiki OS: The Operating System for the Internet of Things. http://www.contiki-os.org/p/about-contiki.html on 1.7.2011

[4] Redwire Redbee Econotag: The Perfect Development board for the Freescale MC13224v ARM7 microcontroller and 802.15.4 radio.
http://www.redwirellc.com/store/node/1 on 1.7.2011

[5] R. Fielding (UC Irvine), J. Gettys (Compaq/W3C), J. Mogul (Compaq), H. Frystyk (W3C/MIT), L. Masinter (Xerox), P. Leach (Microsoft), T. Berners-Lee (W3C/MIT) (1999).. Hypertext Transfer Protocol – HTTP/1.1. *IETF – Network Working Group*
http://www.ietf.org/rfc/rfc2616.txt on 3.2.2012

[6] N. Freed (Innosoft), N. Borenstein (First Virtual) (1996). Multipurpose Internet Mail Extensions. *IETF – Network Working Group*
http://www.mhonarc.org/~ehood/MIME/2045/rfc2045.html on 3.2.2012

[7] Michael Richardson, J. P. Vasseur, Adrian Farrel, Rene Struik, Daniel King. Routing Over Low power and Lossy networks (roll). *IETF-Network Working Group*
http://datatracker.ietf.org/wg/roll/charter/ on 4.2.2012

[8] K. Griffin, J. Rosenberg (Cisco) (2008). Representational State Transfer (REST) for Feature Configuration in Session Initiation Protocol (SIP). *IETF-BLISS*
http://tools.ietf.org/html/draft-griffin-bliss-rest-00#section-1 on 4.2.2012

[9] Z. Shelby (Sensinode), K. Hartke, C. Bormann (Universitaet Bremen TZI), B. Frank (SkyFoundry) (2012). Constrained Application Protocol (CoAP). *IETF-CoRE Working Group*
http://datatracker.ietf.org/doc/draft-ietf-core-coap/ on 13.3.2012

[10] T. Winter (Ed.), P. Thubert (Ed. – Cisco Systems), A. Bradt (Sigma Designs), T. Clausen (LIX, Ecole Polytechnique), J. Hui (Arch Rock Corporation), R. Kelsey (Ember Corporation), P. Levis (Stanford University), K. Pister (Dust Networks), R. Struik, JP. Vasseur (Cisco Systems) (2011). RPL: IPv6 Routing Protocol for Low power and Lossy Networks. *IETF-*

ROLL
http://tools.ietf.org/html/draft-ietf-roll-rpl-19 on 14.3.2012

[11] Mariano Alvira. MC1322x Microcontroller Developers Mailing List http://devl.org/pipermail/mc1322x/ on 5.7.2011

[12] Mariano Alvira. Contiki REST/CoAP Quickstart using Econotags http://mc1322x.devl.org/repos/contiki-mc1322x/cpu/mc1322x/doc/rest-tutorial.md on 5.7.2011

[13] Mariano Alvira. Contiki RPL Quickstart using Econotag http://mc1322x.devl.org/repos/contiki-mc1322x/cpu/mc1322x/doc/rpl-tutorial.md on 6.7.2011

[14] How to split a string on the comma to get an array with 3 strings using C Programming http://cboard.cprogramming.com/c-programming/128736-how-split-string-comma-get-array-3-strings-using-c-programming.html on 20.10.2011

[15] Zolerita (2011). Lesson 1: A first app, knowing Contiki API, timers and blocking conditions http://zolertia.sourceforge.net/wiki/index.php/Mainpage:Contiki_Lesson_1 on 20.10.2011

[16] Zolerita (2011). Lesson 0: Contiki basics, filesystem and compile instructions http://zolertia.sourceforge.net/wiki/index.php/Mainpage:Contiki_Lesson_0 on 20.10.2011

[17] Chee-Yee Chong (IEEE member), Srikanta P. Kumar (Senior IEEE member) (2003). Sensor Networks: Evolution, Opportunities, and Challenges.

[18] Clare, Loren P., Gregory J. Pottie, and Jonathan Agre (1999). "Self-Organizing Distributed Sensor Networks." Proc. SPIE Aerosense99.

[19] Dargie, W. and Poellabauer, C., "Fundamentals of wireless sensor networks: theory and practice", John Wiley and Sons, 2010 ISBN 978-0-470-99765-9, pp. 168–183, 191–192

[20] Sohraby, K., Minoli, D., Znati, T. "Wireless sensor networks: technology, protocols, and applications, John Wiley and Sons", 2007 ISBN 978-0-471-74300-2, pp. 203–209

[21] Garcia P., "A Methodology for the Deployment of Sensor Networks", IEEE Transactions On Knowledge And Data Engineering, vol. 11, no. 4, December 2011.

Appendices

APPENDIX 1 (UDP Client and Sink)

```
/*
 *
 * This file is part of the Contiki operating system.
 * Modified by Edward Sereko Younge, edward.younge@aalto.fi
 */

#include "contiki.h"
#include "lib/random.h"
#include "sys/ctimer.h"
#include "net/uip.h"
#include "net/uip-ds6.h"
#include "net/uip-udp-packet.h"
#include "sys/ctimer.h"
#include "dev/button-sensor.h"
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define UDP_CLIENT_PORT 8888
#define UDP_SERVER_PORT 5555

#define UDP_EXAMPLE_ID  190

#define DEBUG DEBUG_PRINT
#include "net/uip-debug.h"

#ifndef PERIOD
#define PERIOD 60
#endif

#define START_INTERVAL           (15 * CLOCK_SECOND)
#define SEND_INTERVAL      (PERIOD * CLOCK_SECOND)
#define SEND_TIME          (random_rand() % (SEND_INTERVAL))
#define MAX_PAYLOAD_LEN         70

static struct uip_udp_conn *client_conn;
static uip_ipaddr_t server_ipaddr;
char mysnd[80];
char str1[] = "n1 Sim Ends Start 1";
char str2[] = "n1 Sim Ends Start 2";
char str3[] = "n1 Sim Ends Start 3";
char str4[] = "n1 Sim Ends Start 4";
char str5[] = "n1 Sim Ends Start 5";
char str0[100];
```

```
PROCESS(udp_client_process, "UDP client process");
AUTOSTART_PROCESSES(&udp_client_process);
static void
tcpip_handler(void)
{
  char *str;

  if(uip_newdata()) {
    str = uip_appdata;
    str[uip_datalen()] = '\0';
    printf("DATA recv '%s'\n", str);
    sprintf(str0,"%s", str);
    }
}
static void
send_packet(void *ptr)
{
  static int seq_id;
  char buf[MAX_PAYLOAD_LEN];
  static int numds = 5;

  seq_id++;
  PRINTF("DATA send to %d 'Hello %d, this is sim conf and start
msg'\n",
         server_ipaddr.u8[sizeof(server_ipaddr.u8) - 1], seq_id);
    sprintf(buf, "Cnf, 30, Pw, Pi, 5, Si, 2, Tm, Ti, 7, Si, 3, Lg,
Li, 9, Si, 5, %d", numds);
  uip_udp_packet_sendto(client_conn, buf, strlen(buf),
&server_ipaddr, UIP_HTONS(UDP_SERVER_PORT));
}
static void
print_local_addresses(void)
{
  int i;
  uint8_t state;

  PRINTF("Client IPv6 addresses: ");
  for(i = 0; i < UIP_DS6_ADDR_NB; i++) {
    state = uip_ds6_if.addr_list[i].state;
    if(uip_ds6_if.addr_list[i].isused &&
       (state == ADDR_TENTATIVE || state == ADDR_PREFERRED)) {
      PRINT6ADDR(&uip_ds6_if.addr_list[i].ipaddr);
      PRINTF("\n");
      /* hack to make address "final" */
      if (state == ADDR_TENTATIVE) {
     uip_ds6_if.addr_list[i].state = ADDR_PREFERRED;
      }
    }
  }
}
```

```
static void
set_global_address(void)
{
  uip_ipaddr_t ipaddr;

  uip_ip6addr(&ipaddr, 0xaaaa, 0, 0, 0, 0, 0, 0, 0);
  uip_ds6_set_addr_iid(&ipaddr, &uip_lladdr);
  uip_ds6_addr_add(&ipaddr, 0, ADDR_AUTOCONF);

/* The choice of server address determines its 6LoPAN header
compression.
 * (Our address will be compressed Mode 3 since it is derived
from our link-local address)
 * Obviously the choice made here must also be selected in udp-
server.c.
 *
 * For correct Wireshark decoding using a sniffer, add the /64
prefix to the 6LowPAN protocol preferences,
 * e.g. set Context 0 to aaaa::.  At present Wireshark copies
Context/128 and then overwrites it.
 * (Setting Context 0 to aaaa::1111:2222:3333:4444 will report a
16 bit compressed address of aaaa::1111:22ff:fe33:xxxx)
 *
 * Note the IPCMV6 checksum verification depends on the correct
uncompressed addresses.
 */

#if 0
/* Mode 1 - 64 bits inline */
   uip_ip6addr(&server_ipaddr, 0xaaaa, 0, 0, 0, 0, 0, 0, 2);
#elif 1
/* Mode 2 - 16 bits inline */
  uip_ip6addr(&server_ipaddr, 0xaaaa, 0, 0, 0, 0, 0x00ff,
0xfe00, 2);
#else
/* Mode 3 - derived from server link-local (MAC) address */
  uip_ip6addr(&server_ipaddr, 0xaaaa, 0, 0, 0, 0x0250, 0xc2ff,
0xfea8, 0xcd1a); //redbee-econotag
#endif
}
PROCESS_THREAD(udp_client_process, ev, data)
{
  static struct etimer periodic;
//  static struct ctimer backoff_timer;

  PROCESS_BEGIN();

  PROCESS_PAUSE();
```

```
set_global_address();
  SENSORS_ACTIVATE(button_sensor);
  PRINTF("UDP client process started\n");

  print_local_addresses();

  /* new connection with remote host */
  client_conn = udp_new(NULL, UIP_HTONS(UDP_SERVER_PORT), NULL);
  udp_bind(client_conn, UIP_HTONS(UDP_CLIENT_PORT));

  PRINTF("Created a connection with the server ");
  PRINT6ADDR(&client_conn->ripaddr);
  PRINTF(" local/remote port %u/%u\n",
     UIP_HTONS(client_conn->lport), UIP_HTONS(client_conn-
>rport));
  etimer_set(&periodic, CLOCK_SECOND * 10 + random_rand() %
(CLOCK_SECOND * 10));
  while(1) {
     static uint32_t x = 0;
     static uint32_t s1 = 1;
     static int numds2 = 5;
     static uint32_t n1a = 0;
     static uint32_t n1b = 0;
     static uint32_t n1c = 0;
     static uint32_t n1d = 0;

    PROCESS_WAIT_EVENT();
    if(ev == tcpip_event) {
      tcpip_handler();
    }
    else if(etimer_expired(&periodic)) {
      etimer_reset(&periodic);
      if (x < 1){ // If event timer Periodic expires, send
Configuration parameters once
     sprintf(mysnd, "Cnf, 360, Pw, Pi, 5, Si, 2, Tm, Ti, 7, Si,
3, Lg, Li, 9, Si, 5, %d", numds2);
        uip_udp_packet_sendto(client_conn, mysnd, strlen(mysnd),
&server_ipaddr, UIP_HTONS(UDP_SERVER_PORT));
      //ctimer_set(&backoff_timer, SEND_TIME, send_packet, NULL);
      x++;
      }
// If message received is "n1 Sim Ends Start 1" or "n1 Sim Ends
Start 5, send next Configuration parameters plus Start message"
if ((strcmp(str0,str1) == 0) || (strcmp(str0,str5) == 0)){
          //if(n1 == 6){ n1 = 1; printf("n1 reset to 1!\n"); }
          if(n1a < 1){
               sprintf(mysnd, "Start, 360, Pw, Pi, 5, Si, 1,
Tm, Ti, 7, Sm, 3, Lg, Li, 9, Sm, 5, %d", numds2);
```

```
uip_udp_packet_sendto(client_conn, mysnd, strlen(mysnd),
&server_ipaddr, UIP_HTONS(UDP_SERVER_PORT));
                printf("n1's 1st Message Sent!\n");
                free(str0);
                n1a++;
                n1d = 0;
          }
     }
// If message received is "n1 Sim Ends Start 2", send next
Configuration parameters plus Start message"
     if (strcmp(str0,str2) == 0){
          if (n1b < 1){
                        sprintf(mysnd, "Start, 360, Pw, Pi, 5,
Si, 1, Tm, Ti, 7, Sm, 3, Lg, Li, 9, Sm, 5, %d", numds2);
                        uip_udp_packet_sendto(client_conn, mysnd,
strlen(mysnd), &server_ipaddr, UIP_HTONS(UDP_SERVER_PORT));
                        printf("n1's 2nd Message Sent!\n");
                free(str0);
          n1b++;
          }
     }
// If message received is "n1 Sim Ends Start 3", send next
Configuration parameters plus Start message"
     if (strcmp(str0,str3) == 0){
          if (n1c < 1){
                        sprintf(mysnd, "Start, 360, Pw, Pi, 5,
Si, 1, Tm, Ti, 7, Sm, 3, Lg, Li, 9, Sm, 5, %d", numds2);
                        uip_udp_packet_sendto(client_conn, mysnd,
strlen(mysnd), &server_ipaddr, UIP_HTONS(UDP_SERVER_PORT));
                        printf("n1's 3rd Message Sent!\n");
                free(str0);
          n1c++;
          }
     }
// If message received is "n1 Sim Ends Start 4", send next
Configuration parameters plus Start message"
if (strcmp(str0,str4) == 0){
          if (n1d < 1){
                        sprintf(mysnd, "Start, 360, Pw, Pi, 5,
Si, 1, Tm, Ti, 7, Sm, 3, Lg, Li, 9, Sm, 5, %d", numds2);
                        uip_udp_packet_sendto(client_conn, mysnd,
strlen(mysnd), &server_ipaddr, UIP_HTONS(UDP_SERVER_PORT));
                        printf("n1's 4th Message Sent!\n");
                free(str0);
          n1d++;
          n1a = 0;
          n1b = 0;
          n1c = 0;
          } }
    }
```

```
/* Button Sensor Starts ------------------------- */
    else if(ev == sensors_event) {  // If the event was
provoked by the user button, then...
           if(data == &button_sensor) {
           if(s1 == 1){ // If button pressed once, send Start
message
             sprintf(mysnd, "Start, 10, Pw, Pi, 3, Si, 1, Tm,
Ti, 7, Si, 1, Lg, Li, 11, Si, 1, %d", numds2);
             uip_udp_packet_sendto(client_conn, mysnd,
strlen(mysnd), &server_ipaddr, UIP_HTONS(UDP_SERVER_PORT));
             printf(" -- Simulation Start Message Sent! --\n");
           }
           if(s1 == 2){ // If button pressed again, send Stop
message
                  sprintf(mysnd, "Stop, 30, Pw, Pi, 5, Si, 2,
Tm, Ti, 7, Si, 3, Lg, Li, 9, Si, 5, %d", numds2);
                  uip_udp_packet_sendto(client_conn, mysnd,
strlen(mysnd), &server_ipaddr, UIP_HTONS(UDP_SERVER_PORT));
             printf("Simulation Stop Message Sent!\n");
           }
           s1++;
           if(s1 == 4){ // If button pressed agin, send Re-
Start message
                s1 = 2;
                  sprintf(mysnd, "Start, 258, Pw, Pi, 3, Si, 1,
Tm, Ti, 7, Si, 1, Lg, Li, 11, Si, 1, %d", numds2);
                  uip_udp_packet_sendto(client_conn, mysnd,
strlen(mysnd), &server_ipaddr, UIP_HTONS(UDP_SERVER_PORT));
             printf("Simulation Re-Start Message Sent! \n");
           }
           free(mysnd);
           //free(abc_send(&abc));
           }
}
/* Button Sensor Ends ---------------------- */
   else { }
  }

  PROCESS_END();
}
```

APPENDIX 2 (UDP Server)

```
/*
 * This file is part of the Contiki operating system.
 * Modified by Edward Sereko Younge, edward.younge@aalto.fi
 */

#include "contiki.h"
#include "contiki-lib.h"
#include "lib/random.h"
#include "random.h"
#include "contiki-net.h"
#include "net/uip.h"
#include "net/rpl/rpl.h"
#include "rest.h"

#include "net/netstack.h"
#include "dev/button-sensor.h"
#include "dev/leds.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

#define DEBUG DEBUG_PRINT
#include "net/uip-debug.h"

#define UIP_IP_BUF   ((struct uip_ip_hdr
*)&uip_buf[UIP_LLH_LEN])

#define UDP_CLIENT_PORT    8765

#define UDP_SERVER_PORT    5678

#define UDP_EXAMPLE_ID  190
static struct uip_udp_conn *server_conn;

static process_event_t event_data_ready;
char str[100];
char my0[20];
char my1[20];
char my2[20];
char my3[20];
char my4[20];
char my5[20];
char my6[20];
char my7[20];
char my8[20];
char my9[20];
char my10[20];
```

```
char my11[20];
char my12[20];
char my13[20];
char my14[20];
char my15[20];
char my16[20];
char my17[20];
char my62[50];
char my112[50];
char my162[50];
char mycom[] = "End";
char mycom1[] = "Sm";
char all[] = "Cnf";
char mpow[] = " Pw";
char pi[] = " Pi";
char pa[] = " Pa";
char pt[] = " Pt";
char mlite[] = " Lg";
char li[] = " Li";
char la[] = " La";
char lt[] = " Lt";
char mtemp[] = " Tm";
char ti[] = " Ti";
char ta[] = " Ta";
char to[] = " Tt";
char si[] = " Si";
char sa[] = " Sa";
char st[] = " St";
char simstart[] = "Start";
char simstop[] = "Stop";
char str1[50];

char temp[100];

/* Resources are defined by RESOURCE macro, signature: resource
name, the http methods it handles and its url*/
RESOURCE(helloworld, METHOD_GET, "helloworld");
/* For each resource defined, there corresponds an handler
method which should be defined too.
 * Name of the handler method should be [resource name]_handler
 * */
void
helloworld_handler(REQUEST* request, RESPONSE* response)
{
  sprintf(temp,"Hello World n1!\n");
rest_set_header_content_type(response, TEXT_PLAIN);
  rest_set_response_payload(response, (uint8_t*)temp,
strlen(temp));
}
```

```
RESOURCE(discover, METHOD_GET, ".well-known/core");
void
discover_handler(REQUEST* request, RESPONSE* response)
{
  char temp[100];
  int index = 0;
  index += sprintf(temp + index, "%s,",
"</helloworld>;n=\"HelloWorldn1\"");
  rest_set_response_payload(response, (uint8_t*)temp,
strlen(temp));
  rest_set_header_content_type(response,
APPLICATION_LINK_FORMAT);
}
PROCESS(udp_server_process, "UDP server process");
PROCESS(simulation, "The Simulation Process");
AUTOSTART_PROCESSES(&udp_server_process, &simulation);

static void
tcpip_handler(void)
{
  char *appdata;

  if(uip_newdata()) {
    appdata = (char *)uip_appdata;
    appdata[uip_datalen()] = 0;
    PRINTF("DATA recv '%s' from ", appdata);
    PRINTF("%d",
           UIP_IP_BUF->srcipaddr.u8[sizeof(UIP_IP_BUF-
>srcipaddr.u8) - 1]);
    PRINTF("\n");
uip_ipaddr_copy(&server_conn->ripaddr, &UIP_IP_BUF-
>srcipaddr);
int i;
 /* This where the “,” trick was used to separate the
different parts of the received Configuration parameters */
  int count = 0;
  const char delims[] = ",";
  char *result = NULL;
  char **store = NULL;
  char **tmp = NULL;
   sprintf(str, "%s", appdata);
   result = strtok(str, delims);
     if (result != NULL) {
          store = malloc((count + 1) * sizeof(char *));
          store[count] = result;
          count++;
          tmp = malloc(count * sizeof(char *));
          printf("%s\n", result);
          for (i=0; i<count; i++) {
               printf("array is %s\n", store[i]); }
```

```
}
  while (result != NULL) {
       free(tmp);
       tmp = malloc(count * sizeof(char *));
       for (i=0; i<count; i++) {
            tmp[i] = store[i];
       }
       free(store);
       store = malloc((count + 1) * sizeof(char *));
       for (i=0; i<count; i++) {
           store[i] = tmp[i];
       }
       store[count] = result;
       count++;
       printf("%s\n", result);
       for (i=0; i<count; i++) {
            printf("array is %s\n", store[i]);
        if (i == 0){
             sprintf(my0, "%s", store[i]);
        }
             if (i == 1){
                     sprintf(my1, "%s", store[i]);
             }
             if (i == 2){
                     sprintf(my2, "%s", store[i]);
             }
             if (i == 3){
                     sprintf(my3, "%s", store[i]);
             }
             if (i == 4){
                     sprintf(my4, "%s", store[i]);
             }
             if (i == 5){
                     sprintf(my5, "%s", store[i]);
             }
             if (i == 6){
                     sprintf(my6, "%s", store[i]);
             }
             if (i == 7){
                     sprintf(my7, "%s", store[i]);
             }
             if (i == 8){
                     sprintf(my8, "%s", store[i]);
             }
             if (i == 9){
                     sprintf(my9, "%s", store[i]);
             }
            if (i == 10){
                     sprintf(my10, "%s", store[i]);
             }
```

```
            if (i == 11){
                  sprintf(my11, "%s", store[i]);
            }
                  if (i == 12){
                        sprintf(my12, "%s", store[i]);
                  }
                  if (i == 13){
                        sprintf(my13, "%s", store[i]);
                  }
                  if (i == 14){
                        sprintf(my14, "%s", store[i]);
                  }
                  if (i == 15){
                        sprintf(my15, "%s", store[i]);
                  }
                  if (i == 16){
                        sprintf(my16, "%s", store[i]);
                  }
                  if (i == 17){
                        sprintf(my17, "%s", store[i]);
                  }
            }
            result = strtok(NULL, delims);
      }
      free(tmp);
      free(store);
   }
}
/* The comma, ",", trick ends here */

int mp;
int mp1;
int mp2;
int ai;
int bi;
int ci;
/* New Counter Fxn ------------------------------------ */
int ma_counts(){
    int macounts = 0;
    macounts++;
return macounts;
}
/* End Counter fxn ------------------------------------ */
static void
print_local_addresses(void)
{
  int i;
  uint8_t state;

  PRINTF("Server IPv6 addresses: ");
```

```
  for(i = 0; i < UIP_DS6_ADDR_NB; i++) {
    state = uip_ds6_if.addr_list[i].state;
    if(state == ADDR_TENTATIVE || state == ADDR_PREFERRED) {
      PRINT6ADDR(&uip_ds6_if.addr_list[i].ipaddr);
      PRINTF("\n");
      /* hack to make address "final" */
      if (state == ADDR_TENTATIVE) {
     uip_ds6_if.addr_list[i].state = ADDR_PREFERRED;
      }
    }
  }
}
PROCESS_THREAD(udp_server_process, ev, data)
{
  uip_ipaddr_t ipaddr;
  struct uip_ds6_addr *root_if;

  PROCESS_BEGIN();
  //event_data_ready = process_alloc_event();
  PROCESS_PAUSE();
  event_data_ready = process_alloc_event();
  SENSORS_ACTIVATE(button_sensor);

  PRINTF("UDP server started\n");

#if UIP_CONF_ROUTER
/* The choice of server address determines its 6LoPAN header
compression.
 * Obviously the choice made here must also be selected in
udp-client.c.
 *
 * For correct Wireshark decoding using a sniffer, add the /64
prefix to the 6LowPAN protocol preferences,
 * e.g. set Context 0 to aaaa::.  At present Wireshark copies
Context/128 and then overwrites it.
 * (Setting Context 0 to aaaa::1111:2222:3333:4444 will report
a 16 bit compressed address of aaaa::1111:22ff:fe33:xxxx)
 * Note Wireshark's IPCMV6 checksum verification depends on
the correct uncompressed addresses.
 */
#if 0
/* Mode 1 - 64 bits inline */
   uip_ip6addr(&ipaddr, 0xaaaa, 0, 0, 0, 0, 0, 0, 1);
#elif 1
/* Mode 2 - 16 bits inline */
  uip_ip6addr(&ipaddr, 0xaaaa, 0, 0, 0, 0, 0x00ff, 0xfe00, 1);
#else
/* Mode 3 - derived from link local (MAC) address */
  uip_ip6addr(&ipaddr, 0xaaaa, 0, 0, 0, 0, 0, 0, 0);
  uip_ds6_set_addr_iid(&ipaddr, &uip_lladdr);
#endif
```

```
  uip_ds6_addr_add(&ipaddr, 0, ADDR_MANUAL);
  root_if = uip_ds6_addr_lookup(&ipaddr);
  if(root_if != NULL) {
    rpl_dag_t *dag;
    rpl_set_root((uip_ip6addr_t *)&ipaddr);
    dag = rpl_get_dag(RPL_ANY_INSTANCE);
    uip_ip6addr(&ipaddr, 0xaaaa, 0, 0, 0, 0, 0, 0, 0);
    rpl_set_prefix(dag, &ipaddr, 64);
    PRINTF("created a new RPL dag\n");
  } else {
    PRINTF("failed to create a new RPL DAG\n");
  }
#endif /* UIP_CONF_ROUTER */

  print_local_addresses();
  /* The data sink runs with a 100% duty cycle in order to
ensure high
     packet reception rates. */
  NETSTACK_MAC.off(1);

  server_conn = udp_new(NULL, UIP_HTONS(UDP_CLIENT_PORT),
NULL);
  udp_bind(server_conn, UIP_HTONS(UDP_SERVER_PORT));

  PRINTF("Created a server connection with remote address ");
  PRINT6ADDR(&server_conn->ripaddr);
  PRINTF(" local/remote port %u/%u\n", UIP_HTONS(server_conn-
>lport),
         UIP_HTONS(server_conn->rport));
  #ifdef WITH_COAP
  PRINTF("COAP Server in addition to UDP Server running\n");
  #else
  PRINTF("HTTP Server\n");
  #endif

  rest_init();
  rest_activate_resource(&resource_helloworld);
  rest_activate_resource(&resource_discover);
  while(1) {
      static int you;
      static struct etimer et;

      static int sim = 1;
      static int ncont = 1;

    /* Delay 2-4 seconds */
    etimer_set(&et, CLOCK_SECOND * 60 + random_rand() %
(CLOCK_SECOND * 60));
```

```
    PROCESS_YIELD();
    if(ev == tcpip_event) {
      tcpip_handler();
    } else if (ev == sensors_event && data == &button_sensor) {
      PRINTF("Initiaing global repair\n");
      rpl_repair_dag(rpl_get_dag(RPL_ANY_INSTANCE));
    }
   if(etimer_expired(&et)){
    if((strcmp(my0,all) == 0) || (strcmp(my0,simstart) == 0) ||
(strcmp(my0,simstop) == 0)){
     you = atoi(&(*my4));
     process_post(&simulation, event_data_ready, &you);
     printf("Simulation parameters posted!\n");
     if(strcmp(my0,simstop) == 0){ sim = 1; }
    }
    else {
     printf("Waiting for Simulation Input!\n");

     if(strcmp(my0,mycom) == 0){

           ncont++;
          if(ncont == 5){
           if (sim < ((atoi(&(*my17))) + 1)){
/* Sending Done message and requesting for new configuration or
simulation parameters after the current simulation has ended */
               sprintf(str1,"n1 Sim Ends Start %d",sim);

               uip_udp_packet_send(server_conn, str1,
sizeof(str1));
                        sim++;
                        if (sim == ((atoi(&(*my17))) + 1)){ sim
= 2; }

                        strcpy(my0, "Sm");
                ncont = 1;
                }
        }
      }
     }
   }
  }
 PROCESS_END();
}
```

```
/* The SIMULATION PROCESS ------------------------------ */
PROCESS_THREAD(simulation, ev, data)
{
      PROCESS_BEGIN();

      while(1) {
      static uint32_t n1a = 0;
      static uint32_t n1b = 0;
      static uint32_t n1c = 0;
      static uint32_t ticks = 1;
      static uint32_t seconds;
      static uint32_t seconds1;
      static uint32_t seconds2;
      static struct etimer et; // Define etimer 0
      static struct etimer et1; // Define etimer 1
      static struct etimer et2; // Define etimer 2

      PROCESS_WAIT_EVENT();  // Waiting for an event, don't
care which

      if(ev == event_data_ready) {  // If the event was
provoked by the “event_data_ready”, posted from the previous
process
            seconds = (*(uint32_t *)data);
            seconds1 = (uint32_t)(atoi(&(*my9)));
            seconds2 = (uint32_t)(atoi(&(*my14)));

            /* etimers setting */
            etimer_set(&et, CLOCK_SECOND*seconds +
random_rand() % (CLOCK_SECOND * 3));  // Set the timer 0
            etimer_set(&et1, CLOCK_SECOND*seconds1 +
random_rand() % (CLOCK_SECOND * 3));  // Set the timer 1
            etimer_set(&et2, CLOCK_SECOND*seconds2 +
random_rand() % (CLOCK_SECOND * 3));  // Set the timer 2
            printf("+       Timer started        +\n");
         }

      if(etimer_expired(&et)) {  // If the event it's provoked
by the timer expiration, then...
        if(strcmp(my0,simstop) == 0){ ticks = 1; n1a = 0; n1b
= 0; n1c = 0; printf("Simulation Stopped!\n"); } // Stop Sim
with ticks = 10000 and in cmp(my0,conf) if
        if(strcmp(my0,all) == 0){ printf("Configuration
Parameters Received! Waiting for Simulation Start
Message!\n"); }
```

```
/* If Start message is received */
     if(strcmp(my0,simstart) == 0){
       printf("Timer Expires with ticks = %d\n", ticks);
         if(ticks < (atoi(&(*my1)) + 1)){
/* If first quantity is Power, then … */
          if(strcmp(my2,mpow) == 0){
/* If first quantity is Instantaneous Power, then … */
               if(strcmp(my3,pi) == 0){
                    mp1 = ma_counts();
                    if(strcmp(my5,si) == 0){
                         ai = atoi(&(*my6));
                         if(ticks % ai == 0){
                         if(n1a < 1){
                         printf("ai = %d\n", ai);
                         sprintf(my62,"n1 Ipow = %d", mp1);
/* Send Instantaneous Power value */
                        uip_udp_packet_send(server_conn,
my62, sizeof(my62));
                         my62, sizeof(my62));
                         etimer_stop(&et);
                         n1a++;
                              }
                         }
                    }
               }
          }
          n1c = 0;
     } // END eTIMER "et"
          if(etimer_expired(&et1)) {
/* If second quantity is Temperature, then … */
               if(strcmp(my7,mtemp) == 0){
/* If second quantity is Instantaneous Temperature, then … */
                    if(strcmp(my8,ti) == 0){
                    mp2 = ma_counts();
                         if(strcmp(my10,si) == 0){
                              bi = atoi(&(*my11));
                                   if(ticks % bi == 0){
                              if(n1b < 1){
                                  printf("bi = %d\n", bi);
                                   sprintf(my112,"n1 Itmp =
%d", mp2);
/* Send Instantaneous Temperature value */
                                uip_udp_packet_send(server_conn,
my112, sizeof(my112));
                                   etimer_stop(&et1);
                                   n1b++;
                                   }
                                   }
                                   }
                              }}} // END eTIMER "et1"
```

```
           if(etimer_expired(&et2)) {
/* If third quantity is Light, then … */
                if(strcmp(my12,mlite) == 0){
                     //printf("It is Light in my6!\n");
/* If third quantity is Instantaneous Light, then … */
                              if(strcmp(my13,li) == 0){
                                 //READ
                        mp = ma_counts();
                              if(strcmp(my15,si) == 0){
                                   ci = atoi(&(*my16));
                                        if(ticks % ci == 0){
                                          //SEND
                                   if(n1c < 1){
                                   printf("ci = %d\n", ci);
                                              sprintf(my162,"n1
Ilit = %d", mp);

/* Send Instantaneous Light value */
uip_udp_packet_send(server_conn, my162, sizeof(my162));
                                   n1c++;
                                   }
                                   }
                                   }
                              }
                }
/* Restart all etimers to make the combined three periodic */
                etimer_restart(&et);
                etimer_restart(&et1);
                etimer_restart(&et2);
                ticks++;
                n1a = 0; n1b = 0;
           }
           } // ticks comparator or counter ends here

           if (ticks == (atoi(&(*my1)) + 1)){
                printf("Simulation Ended\n");
                n1a = 0; n1b = 0; n1c = 0;
                strcpy(my0, "End");
           }
    if (ticks == (atoi(&(*my1)) + 1)){
/* Reset the periodic counter at end of simulation */
             ticks = 1;
    }
  } // End of Periodic Simulation loop
 } // End of While
 PROCESS_END();
}
/* End Simulation Process */
```

APPENDIX 3 (CoAP Client)

```
/**
 * This file is part of the Contiki operating system.
 * \file
 *      CoAP client example
 * \author
 *      Matthias Kovatsch kovatsch@inf.ethz.ch
 * \modified by
 *              Edward Sereko Younge, edward.younge@aalto.fi
 *
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "contiki.h"
#include "contiki-net.h"

#if !UIP_CONF_IPV6_RPL && !defined
(CONTIKI_TARGET_MINIMAL_NET)
#warning "Compiling with static routing!"
#include "static-routing.h"
#endif

#include "dev/button-sensor.h"

#if WITH_COAP == 3
#include "er-coap-03-engine.h"
#elif WITH_COAP == 6
#include "er-coap-06-engine.h"
#elif WITH_COAP == 7
#include "er-coap-07-engine.h"
#else
#error "CoAP version defined by WITH_COAP not implemented"
#endif


#define DEBUG 0
#if DEBUG
#define PRINTF(...) printf(__VA_ARGS__)
#define PRINT6ADDR(addr)
PRINTF("[%02x%02x:%02x%02x:%02x%02x:%02x%02x:%02x%02x:%02x%02x
:%02x%02x:%02x%02x]", ((u8_t *)addr)[0], ((u8_t *)addr)[1],
((u8_t *)addr)[2], ((u8_t *)addr)[3], ((u8_t *)addr)[4],
((u8_t *)addr)[5], ((u8_t *)addr)[6], ((u8_t *)addr)[7],
((u8_t *)addr)[8], ((u8_t *)addr)[9], ((u8_t *)addr)[10],
((u8_t *)addr)[11], ((u8_t *)addr)[12], ((u8_t *)addr)[13],
((u8_t *)addr)[14], ((u8_t *)addr)[15])
```

```
#define PRINTLLADDR(lladdr)
PRINTF("[%02x:%02x:%02x:%02x:%02x:%02x]",(lladdr)->addr[0],
(lladdr)->addr[1], (lladdr)->addr[2], (lladdr)-
>addr[3],(lladdr)->addr[4], (lladdr)->addr[5])
#else
#define PRINTF(...)
#define PRINT6ADDR(addr)
#define PRINTLLADDR(addr)
#endif

/* TODO: This server address is hard-coded for Cooja. */
//#define SERVER_NODE(ipaddr)   uip_ip6addr(ipaddr, 0xfe80, 0,
0, 0, 0x0212, 0x7402, 0x0002, 0x0202) /* cooja2 */
#define SERVER_NODE(ipaddr)   uip_ip6addr(ipaddr, 0xfe80, 0,
0, 0, 0x0250, 0xc2a8, 0xc012, 0xe31d);

#define LOCAL_PORT      UIP_HTONS(COAP_DEFAULT_PORT+1)
#define REMOTE_PORT     UIP_HTONS(COAP_DEFAULT_PORT)

#define TOGGLE_INTERVAL 200

PROCESS(coap_client_example, "COAP Client Example");
AUTOSTART_PROCESSES(&coap_client_example);

uip_ipaddr_t server_ipaddr[9];
//#define SERVER_NODE(server_ipaddr)
static struct etimer et;

/* Example URIs that can be queried. */
#define NUMBER_OF_URLS 2
//char* service_urls[NUMBER_OF_URLS] = {".well-known/core",
"/toggle", "battery/", "error/in//path"};
char* service_urls[NUMBER_OF_URLS] = {"/hello", "/hello"};
#if PLATFORM_HAS_BUTTON
static int uri_switch = 0;
#endif
/* This function is will be passed to COAP_BLOCKING_REQUEST()
to handle responses. */
void
client_chunk_handler(void *response)
{
  const uint8_t *chunk;
  int len = coap_get_payload(response, &chunk);
  printf("|%.*s", len, (char *)chunk);
}
```

```
PROCESS_THREAD(coap_client_example, ev, data)
{
  PROCESS_BEGIN();

  static coap_packet_t request[1]; /* This way the packet can
be treated as pointer as usual. */
  SERVER_NODE(&server_ipaddr[0]);

  /* receives all CoAP messages */
  coap_receiver_init();

  etimer_set(&et, CLOCK_SECOND / TOGGLE_INTERVAL);

#if PLATFORM_HAS_BUTTON
  SENSORS_ACTIVATE(button_sensor);
  printf("Press a button to request %s\n",
service_urls[uri_switch]);
#endif

  while(1) {
    static int sim2 = 0;
    static int sim3 = 0;
    PROCESS_YIELD();

    if (etimer_expired(&et)) {
      /* defining the IP addresses of the 9 nodes used   */

      uip_ip6addr(&server_ipaddr[0], 0xfe80, 0, 0, 0, 0x0250,
0xc2a8, 0xc012, 0xe31d);
      uip_ip6addr(&server_ipaddr[1], 0xfe80, 0, 0, 0, 0x0250,
0xc2a8, 0xc875, 0xbc57);
      uip_ip6addr(&server_ipaddr[2], 0xfe80, 0, 0, 0, 0x0250,
0xc2a8, 0xcbd1, 0xb3ac);
      uip_ip6addr(&server_ipaddr[3], 0xfe80, 0, 0, 0, 0x0250,
0xc2a8, 0xccd4, 0x21f5);
      uip_ip6addr(&server_ipaddr[4], 0xfe80, 0, 0, 0, 0x0250,
0xc2a8, 0xce5f, 0x3246);
      uip_ip6addr(&server_ipaddr[5], 0xfe80, 0, 0, 0, 0x0250,
0xc2a8, 0xca2d, 0x9b7e);
      uip_ip6addr(&server_ipaddr[6], 0xfe80, 0, 0, 0, 0x0250,
0xc2a8, 0xc694, 0x8b75);
      uip_ip6addr(&server_ipaddr[7], 0xfe80, 0, 0, 0, 0x0250,
0xc2a8, 0xc400, 0x0585);
      uip_ip6addr(&server_ipaddr[8], 0xfe80, 0, 0, 0, 0x0250,
0xc2a8, 0xc119, 0x2938);
```

```
/* send a request to notify the end of the process */
coap_init_message(request, COAP_TYPE_CON, COAP_GET, 0);
      coap_set_header_uri_path(request,
service_urls[uri_switch]);

      printf("--Requesting %s--\n", service_urls[uri_switch]);

      PRINT6ADDR(&server_ipaddr);
      PRINTF(" : %u\n", UIP_HTONS(REMOTE_PORT));

      COAP_BLOCKING_REQUEST(&server_ipaddr[sim3], REMOTE_PORT,
request, client_chunk_handler);
      sim3++;
      if(sim3 == 9){ sim3 = 0;
        }else{ }

#if PLATFORM_HAS_LEDS
/* Defining just two of the nodes, this part not used */
            uip_ip6addr(&server_ipaddr[0], 0xfe80, 0, 0, 0,
0x0250, 0xc2a8, 0xc012, 0xe31d);
      uip_ip6addr(&server_ipaddr[1], 0xfe80, 0, 0, 0, 0x0250,
0xc2a8, 0xc875, 0xbc57);

/* prepare request, TID is set by COAP_BLOCKING_REQUEST() */

      coap_init_message(request, COAP_TYPE_CON, COAP_POST, 0
);
      coap_set_header_uri_path(request, service_urls[1]);
      coap_set_payload(request, (uint8_t *)"Toggle!", 8);
#else

/* Defining just two of the nodes, this part not used */

      uip_ip6addr(&server_ipaddr[0], 0xfe80, 0, 0, 0, 0x0250,
0xc2a8, 0xc012, 0xe31d);
      uip_ip6addr(&server_ipaddr[1], 0xfe80, 0, 0, 0, 0x0250,
0xc2a8, 0xc875, 0xbc57);

/* prepare request, TID is set by COAP_BLOCKING_REQUEST() */

      coap_init_message(request, COAP_TYPE_CON, COAP_GET, 0 );
      coap_set_header_uri_path(request, "hello");
#endif

      PRINT6ADDR(&server_ipaddr);
      PRINTF(" : %u\n", UIP_HTONS(REMOTE_PORT));

      etimer_reset(&et);
```

```
#if PLATFORM_HAS_BUTTON
    } else if (ev == sensors_event && data == &button_sensor)
{

/* Defining just two of the nodes, this part not used */

            uip_ip6addr(&server_ipaddr[0], 0xfe80, 0, 0, 0,
0x0250, 0xc2a8, 0xc012, 0xe31d);
      uip_ip6addr(&server_ipaddr[1], 0xfe80, 0, 0, 0, 0x0250,
0xc2a8, 0xc875, 0xbc57);

/* send a request to notify the end of the process */

      coap_init_message(request, COAP_TYPE_CON, COAP_GET, 0);
      coap_set_header_uri_path(request,
service_urls[uri_switch]);

      printf("--Requesting %s--\n", service_urls[uri_switch]);

      PRINT6ADDR(&server_ipaddr);
      PRINTF(" : %u\n", UIP_HTONS(REMOTE_PORT));

      COAP_BLOCKING_REQUEST(&server_ipaddr[sim2], REMOTE_PORT,
request, client_chunk_handler);
      sim2++;
      if(sim2 == 2){ sim2 = 0;
        }else{ }

      printf("\n--Done--\n");

      uri_switch = (uri_switch+1) % NUMBER_OF_URLS;
#endif

    }
  }

  PROCESS_END();
}
```